A10 Problema do Caminho Mínimo

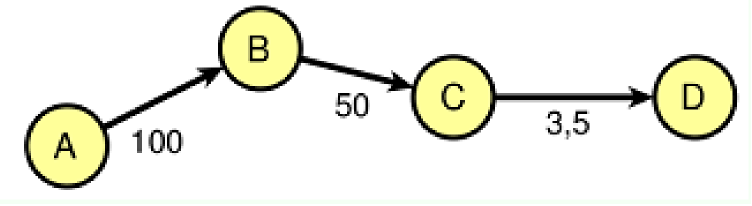
CSI466 – Teoria dos Grafos

Prof. Dr. George H. G. Fonseca
Universidade Federal de Ouro Preto

Introdução

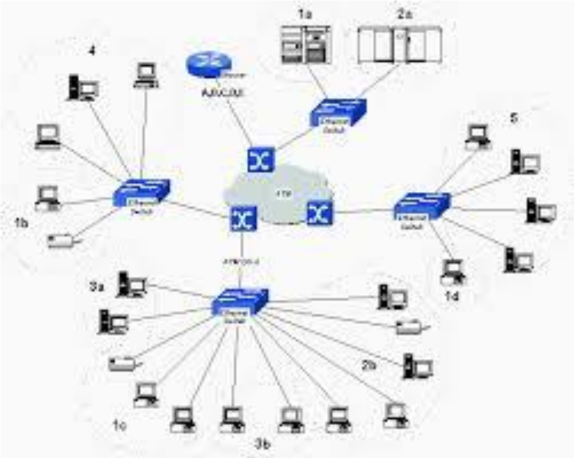
- Considere um grafo G = (V, E) orientado com função peso $w : E \rightarrow R$ que associa cada arco a um valor real $w(u, v)$
  - Obter o caminho de comprimento mínimo entre dois vértices *s* e *t*

- O comprimento de um caminho é a soma dos custos dos arcos que formam o caminho



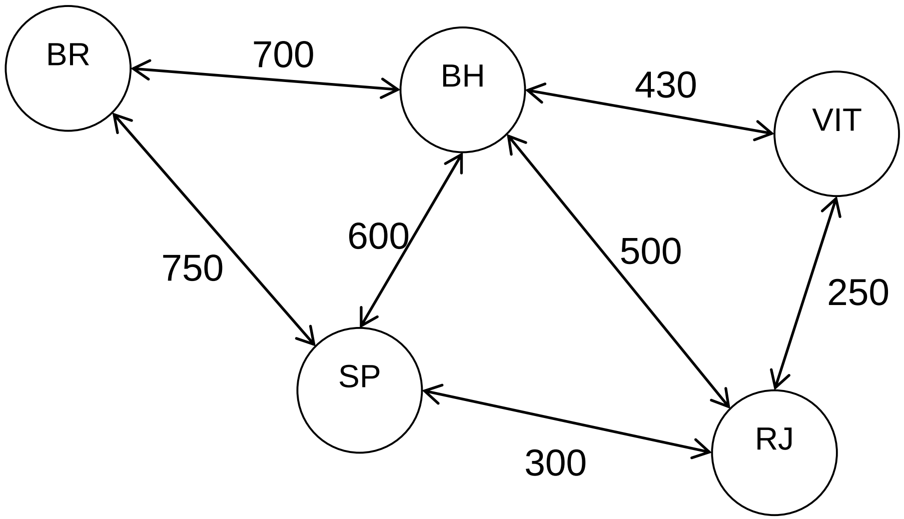
Introdução

- O custo de um arco pode ter várias interpretações de acordo com a aplicação
  - Distâncias
  - Consumo de combustível
  - Tempo
  - Tráfego
  - Custos



Introdução

- Deseja-se encontrar a rota mais rápida de uma cidade A até uma cidade B.
  - Cidades representam vértices
  - Estradas representam arestas
  - Custo de cada aresta indica o tempo necessário para se deslocar pela aresta

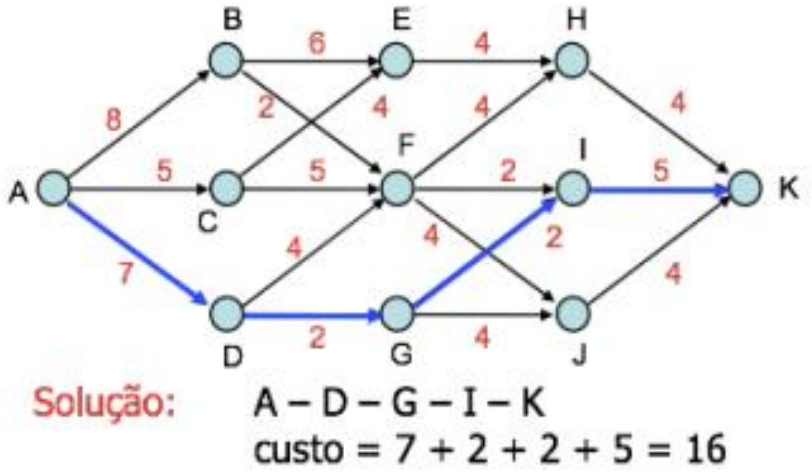


2

Introdução

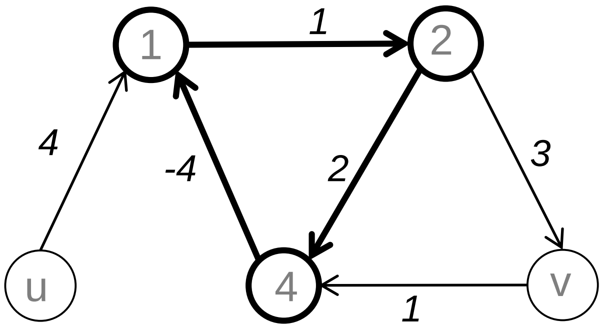
- Construção de estrada entre duas cidades A e K. O grafo abaixo representa os diversos trechos possíveis e o custo de construção de cada um

- Determinar o trajeto ótimo, cujo custo de construção seja mínimo (achar o caminho de menor custo de A a K)



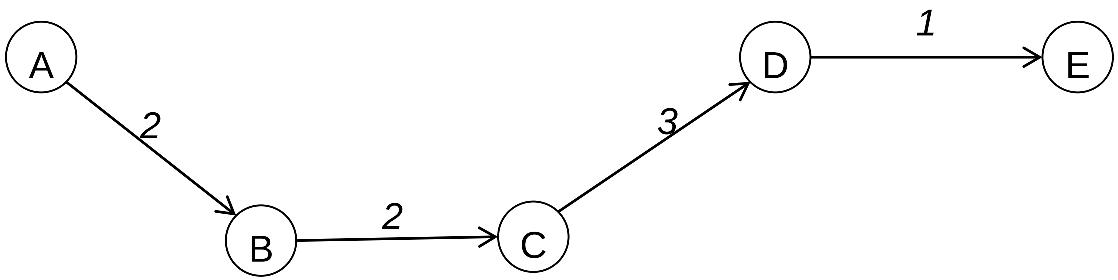
Teoremas

- Condição de existência: para que haja caminho mínimo entre dois vértices u e v, não pode existir no grafo circuito com custo negativo entre os vértices u e v



Teoremas

- Teorema: um subcaminho de um caminho mínimo é também um caminho mínimo



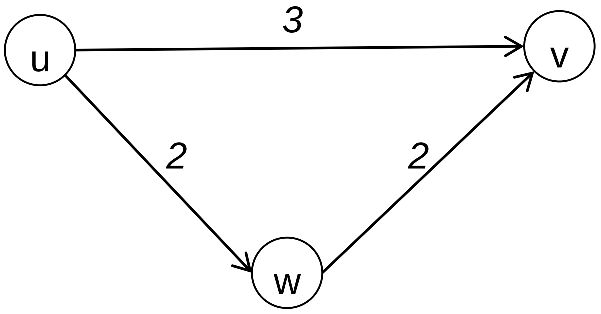
- Assumindo que o caminho entre A e E é mínimo, o caminho entre B e E também é mínimo!

Teoremas

- Teorema (Desigualdade triangular)
  - Uma aresta (u, v) compõe o caminho mínimo se e somente se para todo u, v e w ∈ V:

    $$dist(u, v) \leq dist(u, w) + dist(w, v)$$

  - Demonstração
    - Suponha que não seja verdadeiro. Então, a concatenação dos caminhos mínimos (*u*, *w*) e (*w*, *v*) forma um caminho de *u* a *v* menor do que o caminho mínimo de *u* a *v*.
    - Absurdo



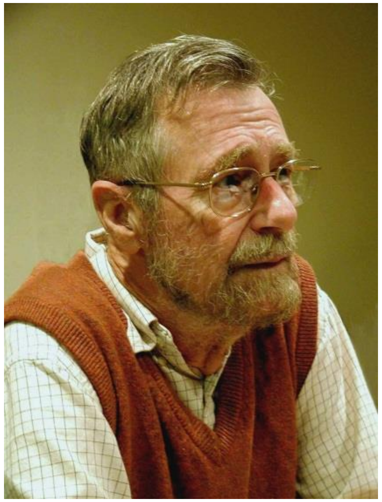
Caminho Mínimo

- As principais variações do problema do caminho mínimo são:
  - **Origem única**: encontrar um caminho mas curto desde uma determinada origem *s* até todo vértice *v*.
  - **Destino único**: encontrar um caminho mas curto até um determinado vértice de destino *t* a partir de cada vértice *v*.
  - **Par único**: encontrar um caminho mais curto de *u* até *v* para determinados vértices *u* e *v*.
  - **Todos pares**: encontrar um caminho mais curto desde *u* até *v*, para todo par de vértices *u* e *v*.

Algoritmo de Dijkstra

- O algoritmo de Dijkstra recebe um grafo orientado G = (V, E, *w*) (sem arestas de peso negativo) e um vértice *s* de G e armazena, para cada vértice v ∊ V, o custo de um caminho mínimo de *s* a *v*.

- Ideia: obter o caminho mínimo para um vértice por iteração até checar todos os vértices



Algoritmo de Dijkstra

- Trabalha com dois vetores:
  - dist[v] distância estimada para cada vértice v
  - pred[v] vértice predecessor ao vértice v no caminho mínimo da estimativa atual

- Abordagem gulosa!
  - Crie um conjunto S de vértices cujas distâncias para o vértice *s* são conhecidas
  - A cada iteração, acrescente a S o vértice $v \in V - S$ cuja distância estimada a *s* é mínima
  - Atualize as distâncias estimadas até *v*

Algoritmo de Dijkstra

```
DIJKSTRA(G(V, E, w), s)          //Parâmetros: representação de grafo (V, E, w)
                                 e vértice origem s
para cada vértice v em V faça
    dist [v] ← ∞                 //dist: vetor que armazena a distância da origem a cada vértice
    pred[v] ← null               //pred: vetor que indica o predecessor de cada vértice no caminho mínimo
                                 a partir da origem
fim-para
dist[s] ← 0
Q ← V                            //Q: conjunto (lista) de vértices a processar (inicialmente contem todos os vértices)
enquanto Q ≠ ∅  faça             //Lista Q não é vazia
    u ← i : min{dist[i], ∀ i ∈ Q}   //u: vértice de menor distância (dist) dentre os vértices de Q
    Q ← Q – {u}                  //Remover vértice u de Q (u foi processado)
    para cada vértice v adjacente a u faça
        se dist[v] > dist[u] + w(u, v) então   //w(u, v): peso da aresta (u, v)
            dist[v] ← dist[u] + w(u, v)
            pred[v] ← u
        fim-se
    fim-para
fim-enquanto
FIM
```

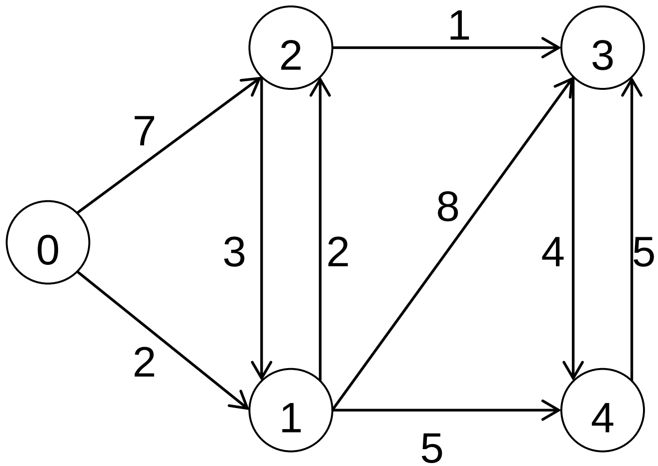
Algoritmo de Dijkstra

```
DIJKSTRA(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    Q ← V
    enquanto Q ≠ ∅  faça
        u ← i : min{dist[i], ∀ i ∈ Q}
        Q ← Q – {u}
        para cada vértice v adjacente a u faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-enquanto
FIM
```

Complexidade de tempo $O(|V|^2)$

*pode ser melhorado!

Algoritmo de Dijkstra



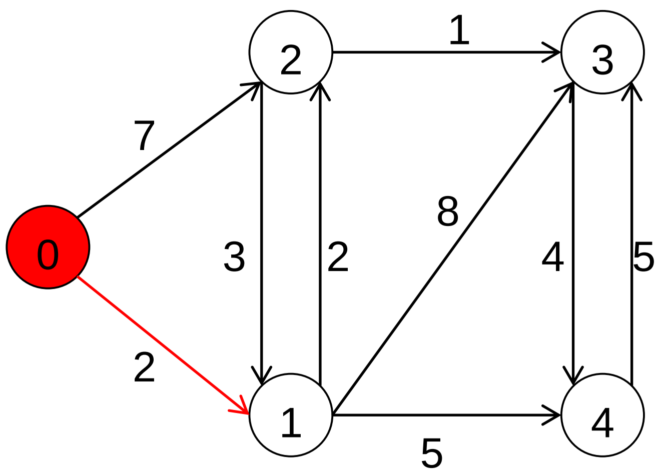

```
DIJKSTRA(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    Q ← V
    enquanto Q ≠ ∅  faça
        u ← i : min{dist[i], ∀ i ∈ Q}
        Q ← Q – {u}
        para cada vértice v adjacente a u faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-enquanto
FIM
```

v	**0**	**1**	**2**	**3**	**4**
dist[v]	0	∞	∞	∞	∞
pred[v]	-	-	-	-	-

s = 0

Q = {0, 1, 2, 3, 4}

Algoritmo de Dijkstra



DIJKSTRA(G(V, E, w), s)

```
para cada vértice v em V faça
    dist [v] ← ∞
    pred[v] ← null
fim-para
dist[s] ← 0
Q ← V
enquanto Q ≠ ∅  faça
    u ← i : min{dist[i], ∀ i ∈ Q}
    Q ← Q – {u}
    para cada vértice v adjacente a u faça
        se dist[v] > dist[u] + w(u, v) então
            dist[v] ← dist[u] + w(u, v)
            pred[v] ← u
        fim-se
    fim-para
fim-enquanto
FIM
```

dist[1] > dist[0] + w(0, 1) ?

$\infty > 0 + 2$

Sim!

Atualize dist[1] e pred[1]

v	**0**	**1**	**2**	**3**	**4**
dist[v]	0	∞	∞	∞	∞
pred[v]	-	-	-	-	-

s = 0

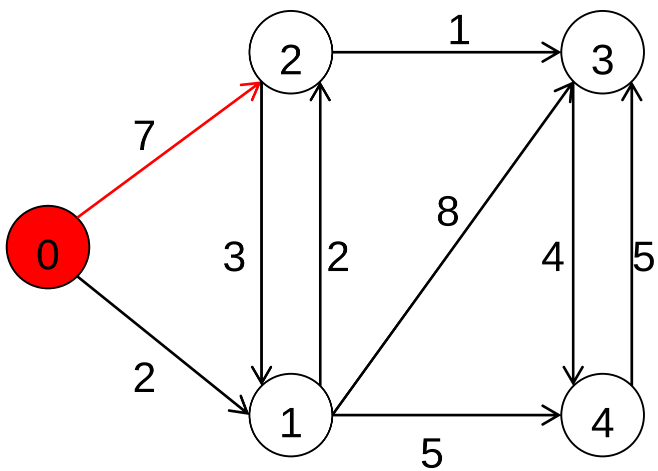
u = 0

v = 1

Q = {1, 2, 3, 4}

Algoritmo de Dijkstra

```
DIJKSTRA(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    Q ← V
    enquanto Q ≠ ∅  faça
        u ← i : min{dist[i], ∀ i ∈ Q}
        Q ← Q – {u}
        para cada vértice v adjacente a u faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-enquanto
FIM
```



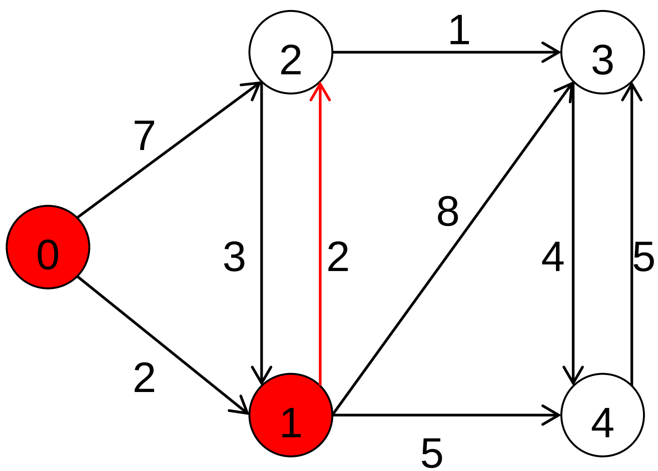
v	0	1	2	3	4
dist[v]	0	2	∞	∞	∞
pred[v]	-	0	-	-	-

dist[2] > dist[0] + w(0, 2) ?
∞ > 0 + 7
Sim!
Atualize dist[2] e pred[2]

s = 0
u = 0
v = 2
Q = {1, 2, 3, 4}

Algoritmo de Dijkstra

```
DIJKSTRA(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    Q ← V
    enquanto Q ≠ ∅  faça
        u ← i : min{dist[i], ∀ i ∈ Q}
        Q ← Q – {u}
        para cada vértice v adjacente a u faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-enquanto
FIM
```



dist[2] > dist[1] + w(1, 2) ?
7 > 2 + 2
Sim!
Atualize dist[2] e pred[2]

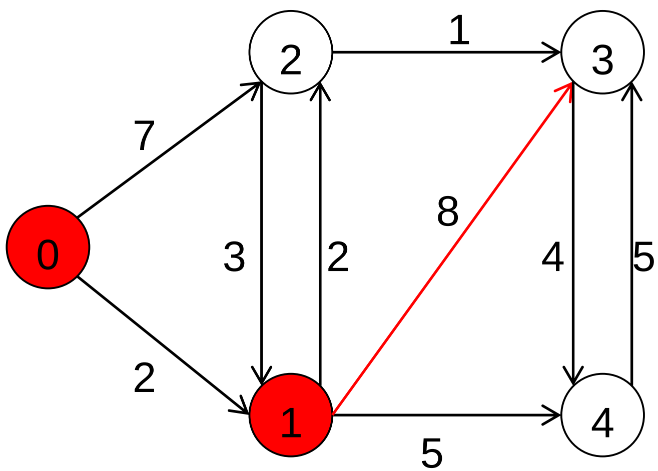
v	0	1	2	3	4
dist[v]	0	2	7	∞	∞
pred[v]	-	0	0	-	-

s = 0
u = 1
v = 2
Q = {2, 3, 4}

Algoritmo de Dijkstra

DIJKSTRA(G(V, E, w), s)

para cada vértice v em V **faça**
  dist [v] ← ∞
  pred[v] ← null
fim-para
dist[s] ← 0
Q ← V
enquanto Q ≠ ∅ faça
  u ← i : min{dist[i], ∀ i ∈ Q}
  Q ← Q – {u}
  para cada vértice v adjacente a u **faça**
    se dist[v] > dist[u] + w(u, v) **então**
      dist[v] ← dist[u] + w(u, v)
      pred[v] ← u
    fim-se
  fim-para
fim-enquanto
FIM



v	**0**	**1**	**2**	**3**	**4**
dist[v]	0	2	4	∞	∞
pred[v]	-	0	1	-	-

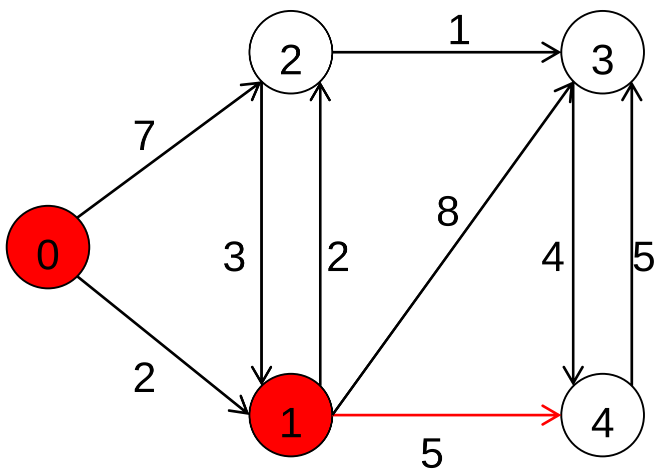
dist[3] > dist[1] + w(1, 3) ?
∞ > 2 + 8
Sim!
Atualize dist[3] e pred[3]

s = 0
u = 1
v = 3
Q = {2, 3, 4}

Algoritmo de Dijkstra

DIJKSTRA(G(V, E, w), s)

para cada vértice v em V **faça**
    dist [v] ← ∞
    pred[v] ← null
fim-para
dist[s] ← 0
Q ← V
enquanto Q ≠ ∅ faça
    u ← i : min{dist[i], ∀ i ∈ Q}
    Q ← Q – {u}
    para cada vértice v adjacente a u **faça**
        se dist[v] > dist[u] + w(u, v) **então**
            dist[v] ← dist[u] + w(u, v)
            pred[v] ← u
        fim-se
    fim-para
fim-enquanto
FIM



dist[4] > dist[1] + w(1, 4) ?
∞ > 2 + 5
Sim!
Atualize dist[4] e pred[4]

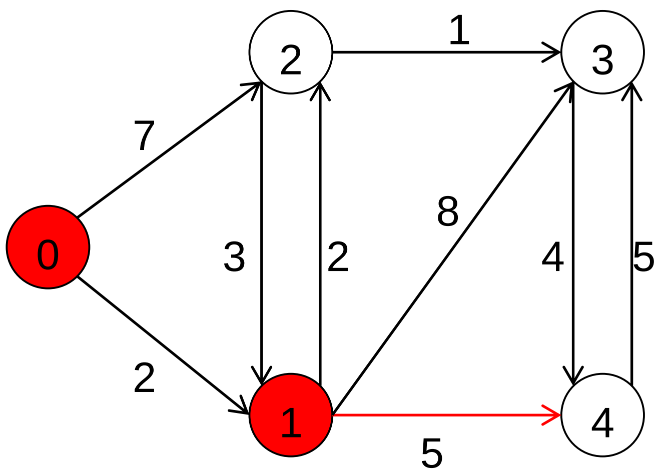
v	**0**	**1**	**2**	**3**	**4**
dist[v]	0	2	4	10	∞
pred[v]	-	0	1	1	-

s = 0
u = 1
v = 3
Q = {2, 3, 4}

Algoritmo de Dijkstra

DIJKSTRA(G(V, E, w), s)

para cada vértice v em V **faça**
    dist [v] ← ∞
    pred[v] ← null
fim-para
dist[s] ← 0
Q ← V
enquanto Q ≠ ∅ faça
    u ← i : min{dist[i], ∀ i ∈ Q}
    Q ← Q – {u}
    para cada vértice v adjacente a u **faça**
        se dist[v] > dist[u] + w(u, v) **então**
            dist[v] ← dist[u] + w(u, v)
            pred[v] ← u
        fim-se
    fim-para
fim-enquanto
FIM



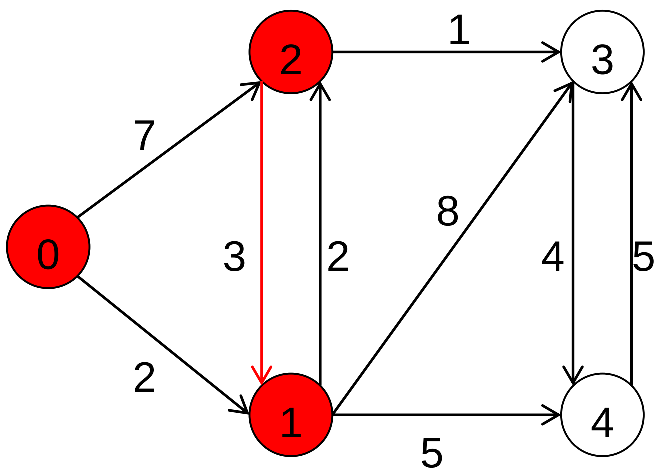
dist[4] > dist[1] + w(1, 4) ?
∞ > 2 + 5
Sim!
Atualize dist[4] e pred[4]

v	**0**	**1**	**2**	**3**	**4**
dist[v]	0	2	4	10	7
pred[v]	-	0	1	1	1

s = 0
u = 1
v = 3
Q = {2, 3, 4}

Algoritmo de Dijkstra

```
DIJKSTRA(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    Q ← V
    enquanto Q ≠ Ø  faça
        u ← i : min{dist[i], ∀ i ∈ Q}
        Q ← Q – {u}
        para cada vértice v adjacente a u faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-enquanto
FIM
```



dist[1] > dist[2] + w(2, 1) ?

2 > 4 + 3

Não!

v	0	1	2	3	4
dist[v]	0	2	4	10	7
pred[v]	-	0	1	1	1

s = 0

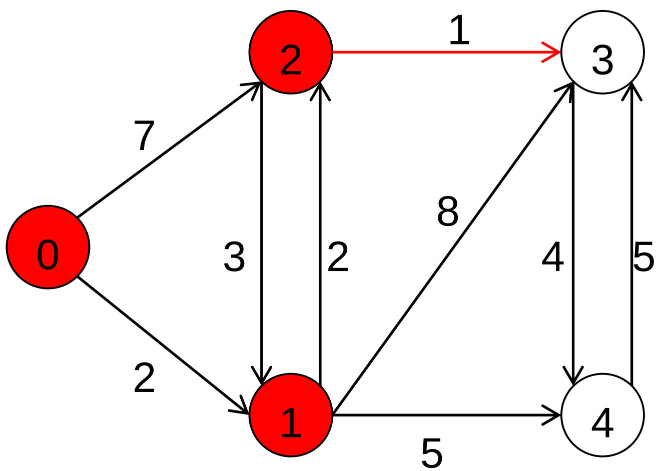
u = 2

v = 1

Q = {3, 4}

Algoritmo de Dijkstra

```
DIJKSTRA(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    Q ← V
    enquanto Q ≠ ∅  faça
        u ← i : min{dist[i], ∀ i ∈ Q}
        Q ← Q – {u}
        para cada vértice v adjacente a u faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-enquanto
FIM
```



dist[3] > dist[2] + w(2, 3) ?

10 > 4 + 1

Sim!

Atualize dist[3] e pred[3]

v	0	1	2	3	4
dist[v]	0	2	4	10	7
pred[v]	-	0	1	1	1

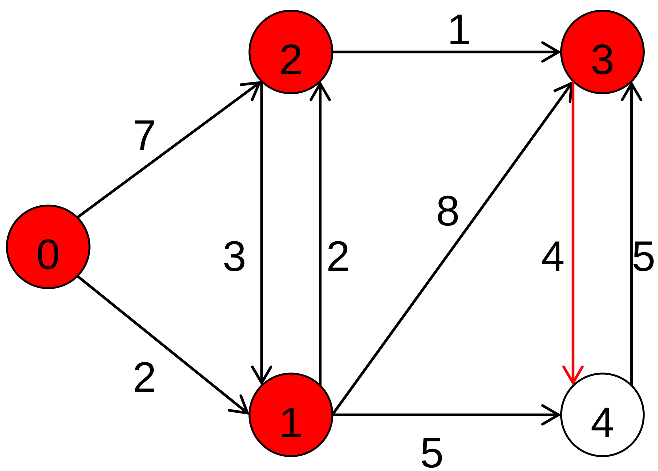
s = 0

u = 2

v = 1

Q = {3, 4}

Algoritmo de Dijkstra




```
DIJKSTRA(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    Q ← V
    enquanto Q ≠ ∅  faça
        u ← i : min{dist[i], ∀ i ∈ Q}
        Q ← Q – {u}
        para cada vértice v adjacente a u faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-enquanto
FIM
```

dist[4] > dist[3] + w(3, 4) ?

7 > 5 + 4

Não!

v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

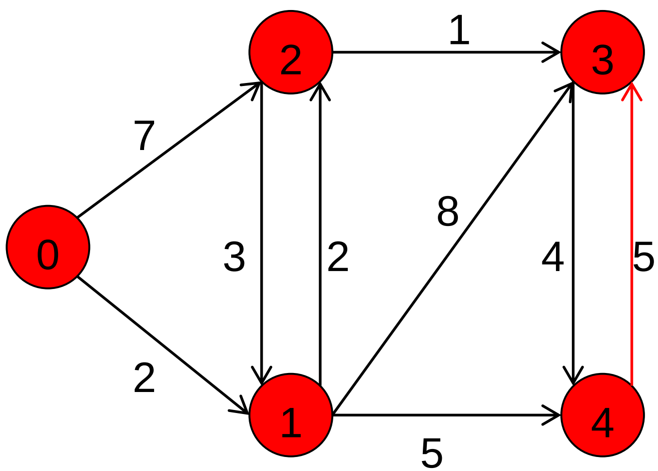
s = 0

u = 3

v = 4

Q = {4}

Algoritmo de Dijkstra




```
DIJKSTRA(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    Q ← V
    enquanto Q ≠ Ø  faça
        u ← i : min{dist[i], ∀ i ∈ Q}
        Q ← Q – {u}
        para cada vértice v adjacente a u faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-enquanto
FIM
```

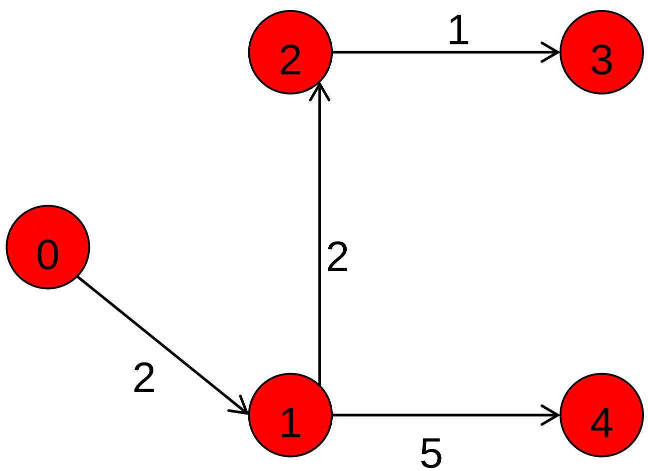
dist[3] > dist[4] + w(4, 3) ?
5 > 7 + 5
Não!

v	**0**	**1**	**2**	**3**	**4**
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0
u = 4
v = 3
Q = {}

Algoritmo de Dijkstra

Arestas dos caminhos mínimos:



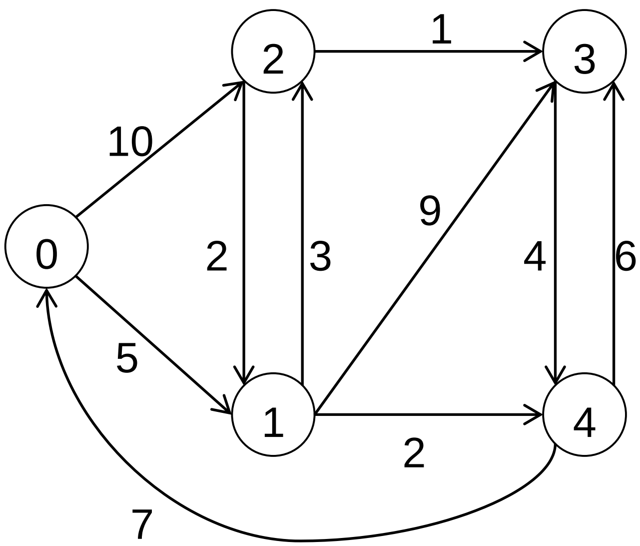

```
DIJKSTRA(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    Q ← V
    enquanto Q ≠ ∅  faça
        u ← i : min{dist[i], ∀ i ∈ Q}
        Q ← Q – {u}
        para cada vértice v adjacente a u faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-enquanto
FIM
```

v	**0**	**1**	**2**	**3**	**4**
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0
u = 4
v = 3
Q = {}   Fim do algoritmo!

Algoritmo de Dijkstra
Exercício

- Simule a execução do algoritmo de Dijkstra para o grafo abaixo



Algoritmo de Bellman-Ford

- Permite arestas de peso negativo

- Verifica a existência de ciclo de peso negativo a partir da origem
  - Se existir, indica que não é possível achar menor caminho

- Ideia: avaliar aresta a aresta para progressivamente diminuir as estimativas de distância até encontrar o menor caminho

Algoritmo de Bellman-Ford

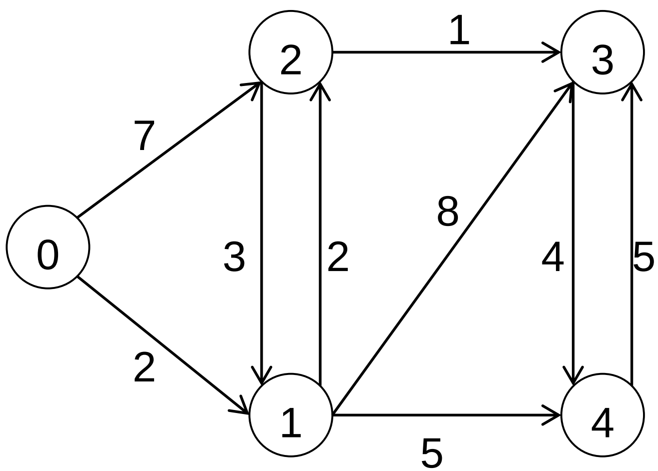
```
BELLMAN-FORD(G(V, E, w), s)          //Parâmetros: representação de grafo (V, E, w)
                                     e vértice origem s
    para cada vértice v em V faça
        dist [v] ← ∞                 //dist: vetor que armazena a distância da origem a cada vértice
        pred[v] ← null               //pred: vetor que indica o predecessor de cada vértice no caminho mínimo
                                     a partir da origem
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então   //w(u, v): peso da aresta (u, v)
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE            //há ciclo de custo negativo
        fim-se
    fim-para
    retorne TRUE
FIM
```

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
para cada vértice v em V faça
    dist [v] ← ∞
    pred[v] ← null
fim-para
dist[s] ← 0
para cada vértice i em V faça
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            dist[v] ← dist[u] + w(u, v)
            pred[v] ← u
        fim-se
    fim-para
fim-para
para cada aresta (u, v) em E faça
    se dist[v] > dist[u] + w(u, v) então
        retorne FALSE
    fim-se
fim-para
retorne TRUE
FIM
```

Complexidade de tempo O(|V| x |E|)

Algoritmo de Bellman-Ford




```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
FIM
```

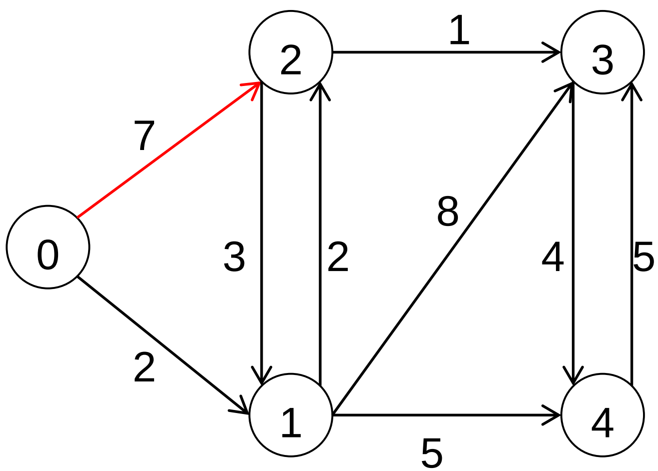
v	0	1	2	3	4
dist[v]	0	∞	∞	∞	∞
pred[v]	-	-	-	-	-

s = 0

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
FIM
```



dist[2] > dist[0] + w(0, 2) ?

$\infty > 0 + 7$

Sim!

Atualize dist[2] e pred[2]

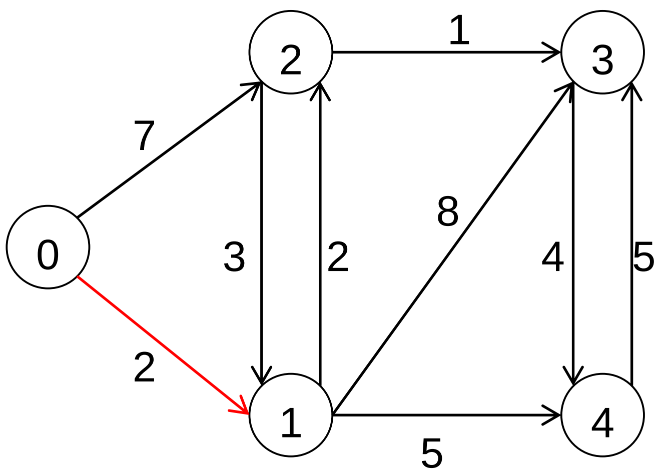
v	0	1	2	3	4
dist[v]	0	∞	7	∞	∞
pred[v]	-	-	0	-	-

s = 0
i = 0
u = 0
v = 2

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
Fim
```



dist[1] > dist[0] + w(0, 1) ?

$\infty > 0 + 2$

Sim!

Atualize dist[1] e pred[1]

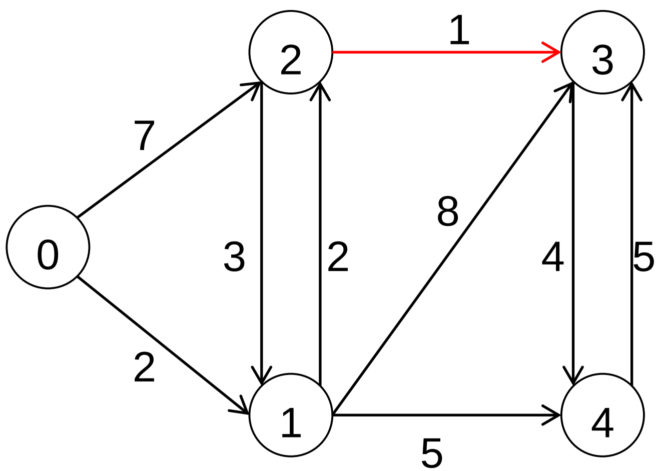
v	0	1	2	3	4
dist[v]	0	2	7	∞	∞
pred[v]	-	0	0	-	-

s = 0
i = 0
u = 0
v = 2

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
FIM
```



dist[3] > dist[2] + w(2, 3) ?

$\infty > 7 + 1$

Sim!

Atualize dist[3] e pred[3]

v	0	1	2	3	4
dist[v]	0	2	7	8	∞
pred[v]	-	0	0	2	-

s = 0

i = 0

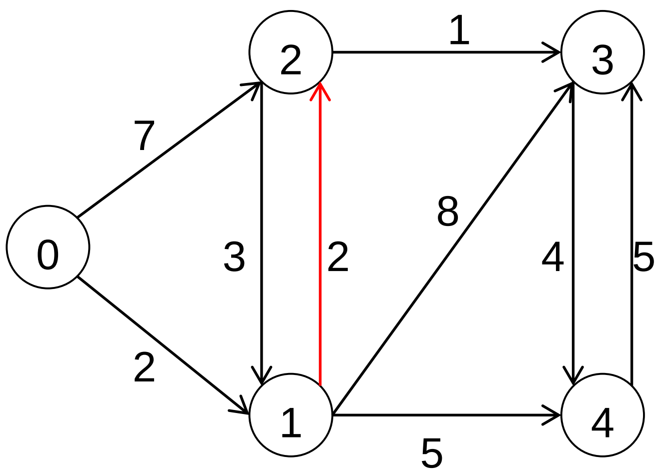
u = 2

v = 3

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
Fim
```



dist[2] > dist[1] + w(1, 2) ?
7 > 2 + 2
Sim!
Atualize dist[2] e pred[2]

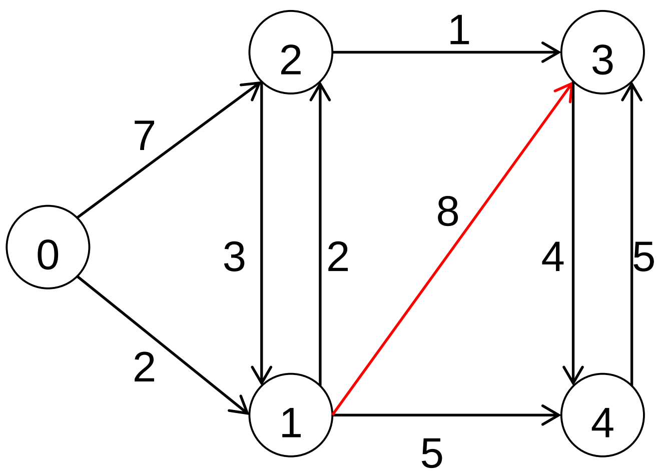
v	0	1	2	3	4
dist[v]	0	2	4	8	∞
pred[v]	-	0	1	2	-

s = 0
i = 0
u = 1
v = 2

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
FIM
```



dist[3] > dist[1] + w(1, 3) ?

8 > 2 + 8

Não!

v	0	1	2	3	4
dist[v]	0	2	4	8	∞
pred[v]	-	0	1	2	-

s = 0

i = 0

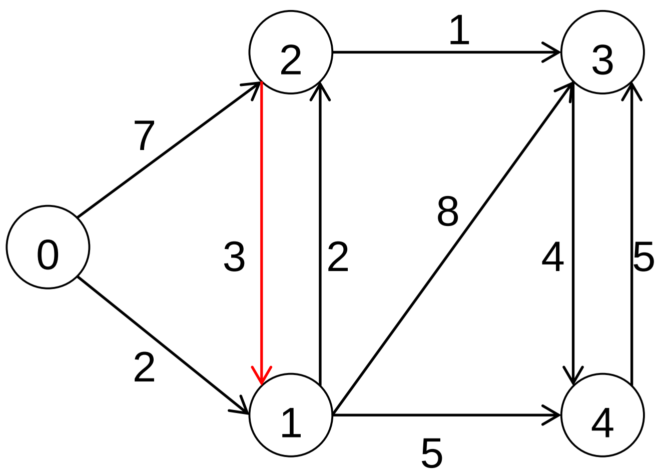
u = 1

v = 3

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
FIM
```



dist[1] > dist[2] + w(2, 1) ?
2 > 2 + 3
Não!

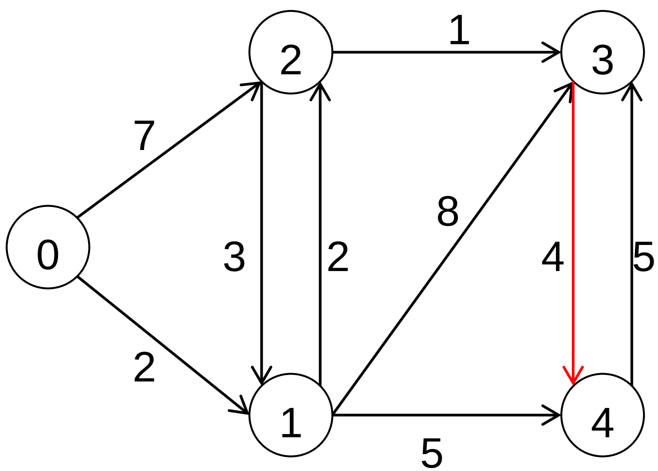
v	**0**	**1**	**2**	**3**	**4**
dist[v]	0	2	4	8	∞
pred[v]	-	0	1	2	-

s = 0
i = 0
u = 2
v = 1

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
FIM
```



dist[4] > dist[3] + w(3, 4) ?

$\infty > 8 + 4$

Sim!

Atualize dist[4] e pred[4]

v	0	1	2	3	4
dist[v]	0	2	4	8	12
pred[v]	-	0	1	2	3

s = 0

i = 0

u = 3

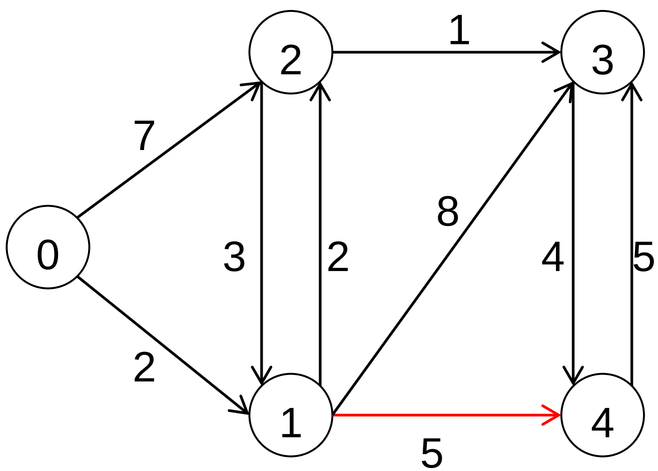
v = 4

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

BELLMAN-FORD(G(V, E, w), s)

```
para cada vértice v em V faça
    dist [v] ← ∞
    pred[v] ← null
fim-para
dist[s] ← 0
para cada vértice i em V faça
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            dist[v] ← dist[u] + w(u, v)
            pred[v] ← u
        fim-se
    fim-para
fim-para
para cada aresta (u, v) em E faça
    se dist[v] > dist[u] + w(u, v) então
        retorne FALSE
    fim-se
fim-para
retorne TRUE
FIM
```



dist[4] > dist[1] + w(1, 4) ?

12 > 2 + 5

Sim!

Atualize dist[4] e pred[4]

v	0	1	2	3	4
dist[v]	0	2	4	8	7
pred[v]	-	0	1	2	1

s = 0

i = 0

u = 1

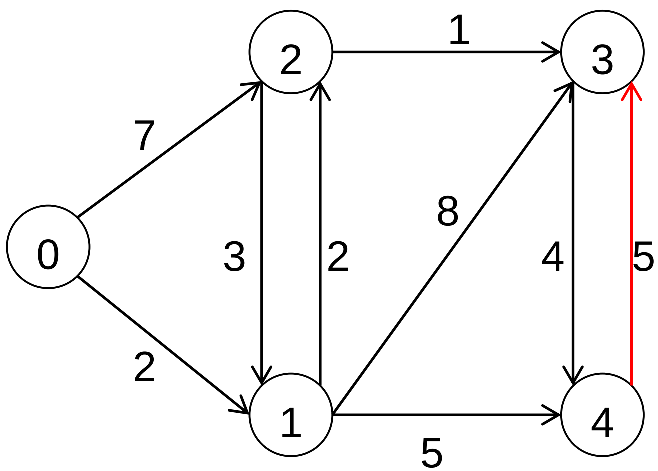
v = 4

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

BELLMAN-FORD(G(V, E, w), s)

```
para cada vértice v em V faça
    dist [v] ← ∞
    pred[v] ← null
fim-para
dist[s] ← 0
para cada vértice i em V faça
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            dist[v] ← dist[u] + w(u, v)
            pred[v] ← u
        fim-se
    fim-para
fim-para
para cada aresta (u, v) em E faça
    se dist[v] > dist[u] + w(u, v) então
        retorne FALSE
    fim-se
fim-para
retorne TRUE
FIM
```



dist[3] > dist[4] + w(4, 3) ?

8 > 7 + 5

Não!

v	0	1	2	3	4
dist[v]	0	2	4	8	7
pred[v]	-	0	1	2	1

s = 0

i = 0

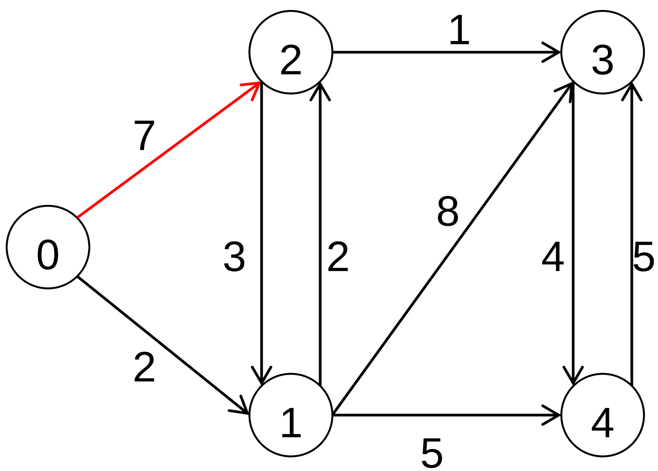
u = 4

v = 3

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
FIM
```



dist[2] > dist[0] + w(0, 2) ?

4 > 0 + 7

Não!

v	0	1	2	3	4
dist[v]	0	2	4	8	7
pred[v]	-	0	1	2	1

s = 0

i = ~~0~~ 1

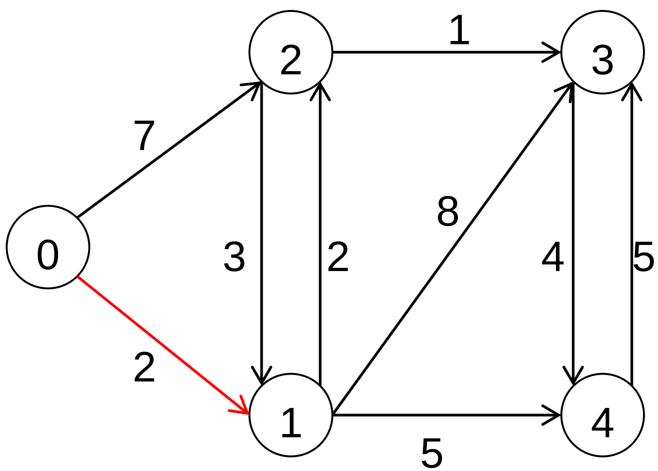
u = 0

v = 2

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
FIM
```



dist[1] > dist[0] + w(0, 1) ?
2 > 0 + 2
Não!

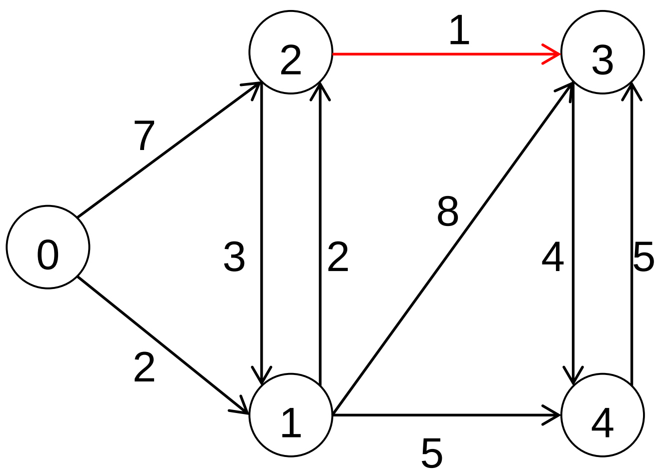
v	**0**	**1**	**2**	**3**	**4**
dist[v]	0	2	4	8	7
pred[v]	-	0	1	2	1

s = 0
i = ~~0~~ 1
u = 0
v = 1

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
FIM
```



dist[3] > dist[2] + w(2, 3) ?

8 > 4 + 1

Sim!

Atualize dist[3] e pred[3]

v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0

i = ~~0~~ 1

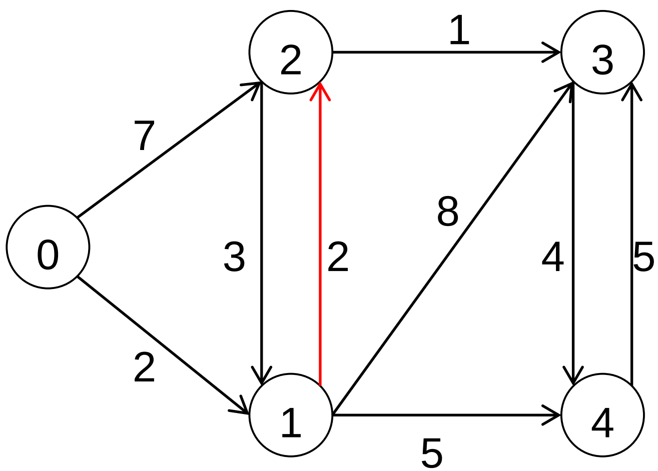
u = 2

v = 3

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
para cada vértice v em V faça
    dist [v] ← ∞
    pred[v] ← null
fim-para
dist[s] ← 0
para cada vértice i em V faça
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            dist[v] ← dist[u] + w(u, v)
            pred[v] ← u
        fim-se
    fim-para
fim-para
para cada aresta (u, v) em E faça
    se dist[v] > dist[u] + w(u, v) então
        retorne FALSE
    fim-se
fim-para
retorne TRUE
FIM
```



dist[2] > dist[1] + w(1, 2) ?

4 > 2 + 2

Não!

v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0

i = ~~0~~ 1

u = 1

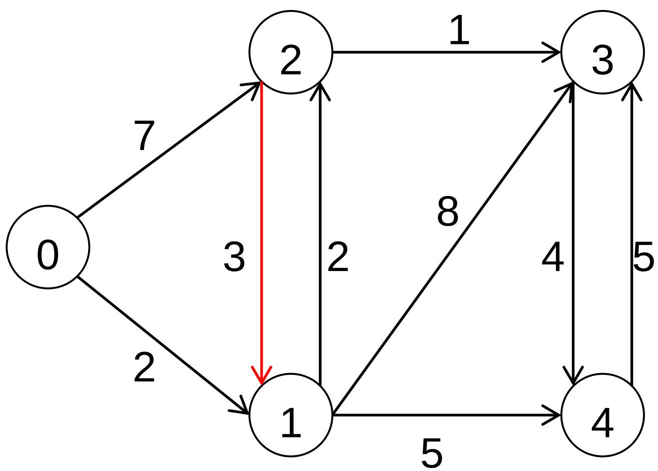
v = 2

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

BELLMAN-FORD(G(V, E, w), s)

```
para cada vértice v em V faça
    dist [v] ← ∞
    pred[v] ← null
fim-para
dist[s] ← 0
para cada vértice i em V faça
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            dist[v] ← dist[u] + w(u, v)
            pred[v] ← u
        fim-se
    fim-para
fim-para
para cada aresta (u, v) em E faça
    se dist[v] > dist[u] + w(u, v) então
        retorne FALSE
    fim-se
fim-para
retorne TRUE
Fim
```



dist[1] > dist[2] + w(2, 1) ?

2 > 4 + 3

Não!

v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0

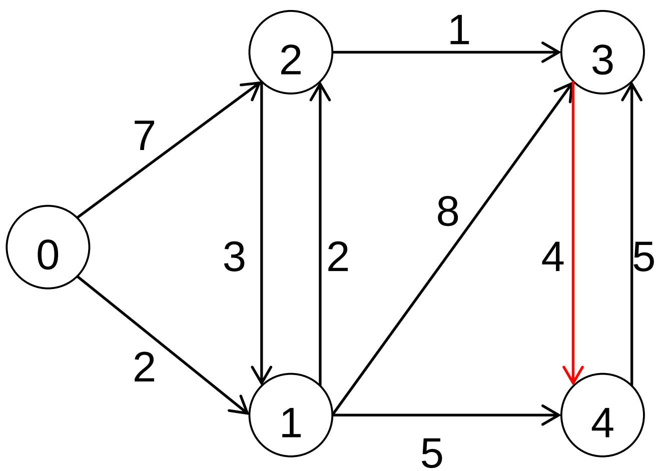
i = ~~0~~ 1

u = 2

v = 1

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford




```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
FIM
```

dist[4] > dist[3] + w(3, 4) ?
7 > 5 + 4
Não!

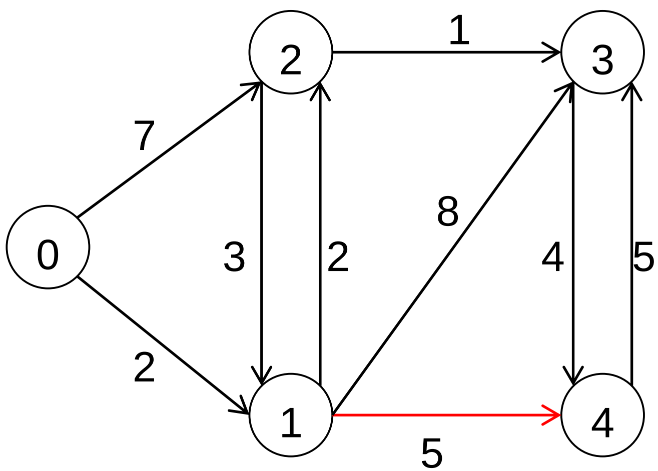
v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0
i = ~~0~~ 1
u = 3
v = 4

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
para cada vértice v em V faça
    dist [v] ← ∞
    pred[v] ← null
fim-para
dist[s] ← 0
para cada vértice i em V faça
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            dist[v] ← dist[u] + w(u, v)
            pred[v] ← u
        fim-se
    fim-para
fim-para
para cada aresta (u, v) em E faça
    se dist[v] > dist[u] + w(u, v) então
        retorne FALSE
    fim-se
fim-para
retorne TRUE
Fim
```



dist[4] > dist[1] + w(1, 4) ?

7 > 2 + 5

Não!

v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0

i = ~~0~~ 1

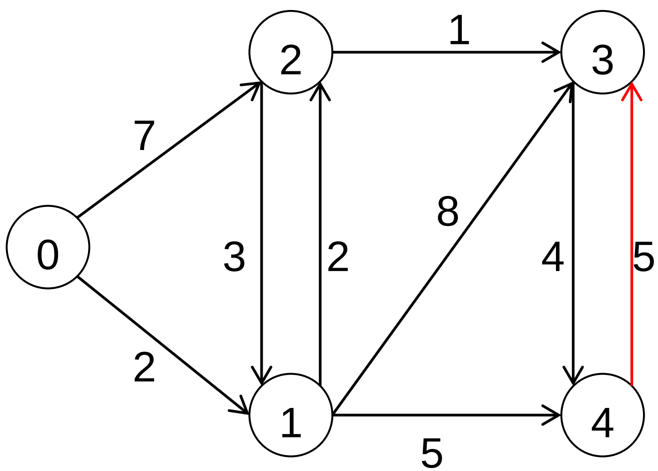
u = 1

v = 4

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
FIM
```



dist[3] > dist[4] + w(4, 3) ?

5 > 7 + 5

Não!

v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0

i = ~~0~~ 1

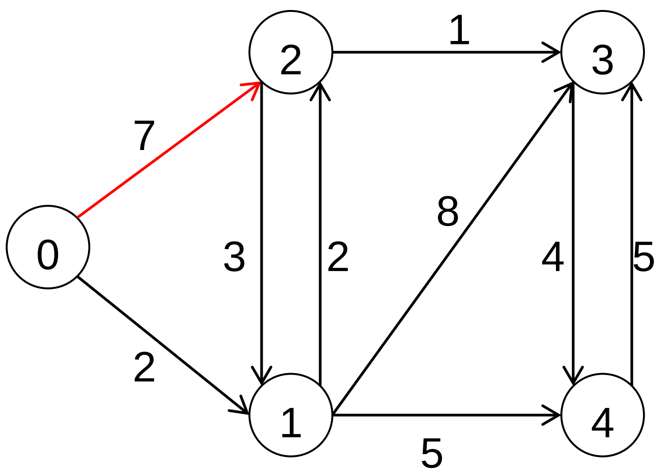
u = 4

v = 3

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
FIM
```



dist[2] > dist[0] + w(0, 2) ?

4 > 0 + 7

Não!

v	0	1	2	3	4
dist[v]	0	2	4	8	7
pred[v]	-	0	1	2	1

s = 0

i = ~~0~~ ~~1~~ 2

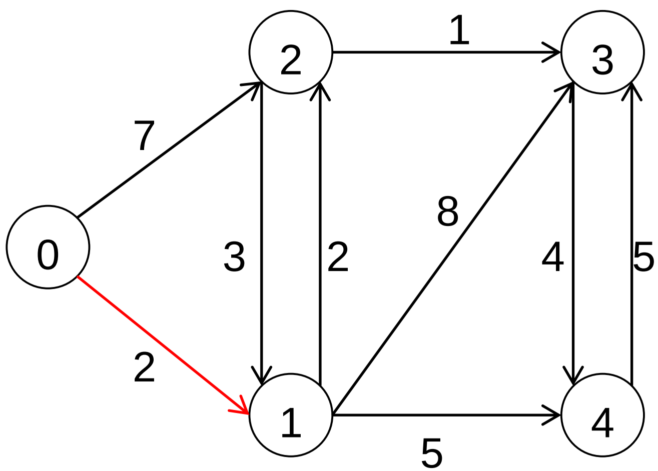
u = 0

v = 2

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
Fim
```



dist[1] > dist[0] + w(0, 1) ?
2 > 0 + 2
Não!

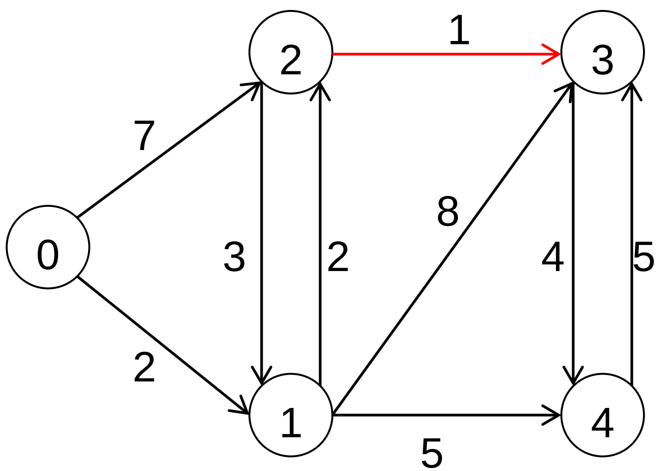
v	0	1	2	3	4
dist[v]	0	2	4	8	7
pred[v]	-	0	1	2	1

s = 0
i = ~~0~~ ~~1~~ 2
u = 0
v = 1

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
       se dist[v] > dist[u] + w(u, v) então
           retorne FALSE
        fim-se
    fim-para
    retorne TRUE
FIM
```



dist[3] > dist[2] + w(2, 3) ?

5 > 4 + 1

Não!

v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0

i = ~~0~~ ~~1~~ 2

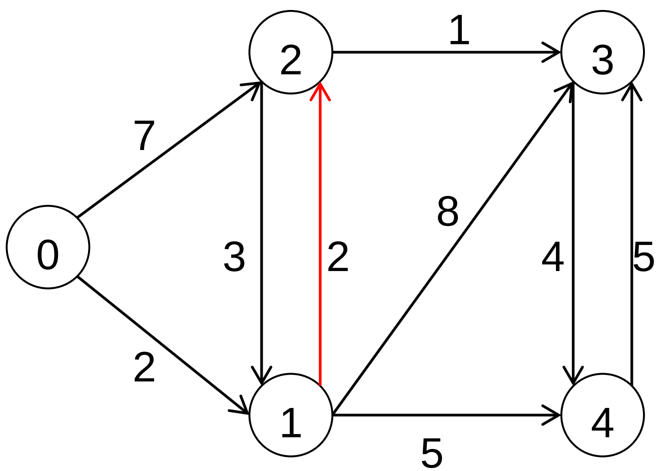
u = 2

v = 3

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
FIM
```



dist[2] > dist[1] + w(1, 2) ?

4 > 2 + 2

Não!

v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0

i = ~~0~~ ~~1~~ 2

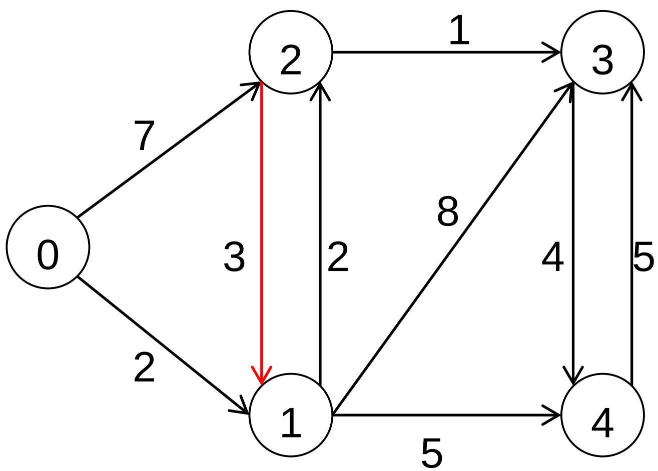
u = 1

v = 2

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
FIM
```



dist[1] > dist[2] + w(2, 1) ?

2 > 4 + 3

Não!

v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0

i = ~~0~~ ~~1~~ 2

u = 2

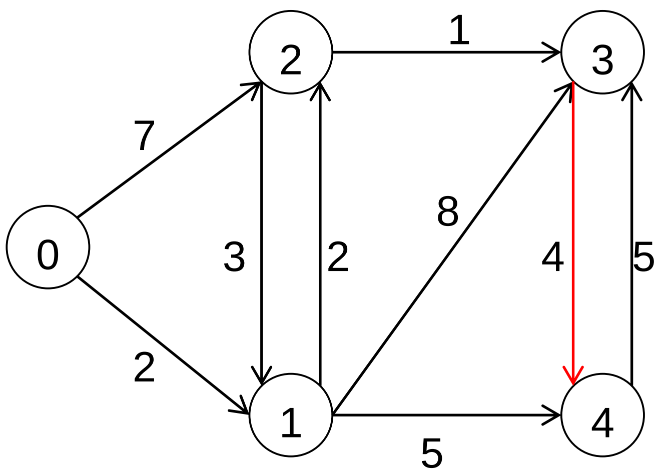
v = 1

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

BELLMAN-FORD(G(V, E, w), s)

para cada vértice v em V **faça**
    dist [v] ← ∞
    pred[v] ← null
fim-para
dist[s] ← 0
para cada vértice i em V **faça**
    para cada aresta (u, v) em E **faça**
        se dist[v] > dist[u] + w(u, v) **então**
            dist[v] ← dist[u] + w(u, v)
            pred[v] ← u
        fim-se
    fim-para
fim-para
para cada aresta (u, v) em E **faça**
    se dist[v] > dist[u] + w(u, v) **então**
        retorne FALSE
    fim-se
fim-para
retorne TRUE
FIM



dist[4] > dist[3] + w(3, 4) ?
7 > 5 + 4
Não!

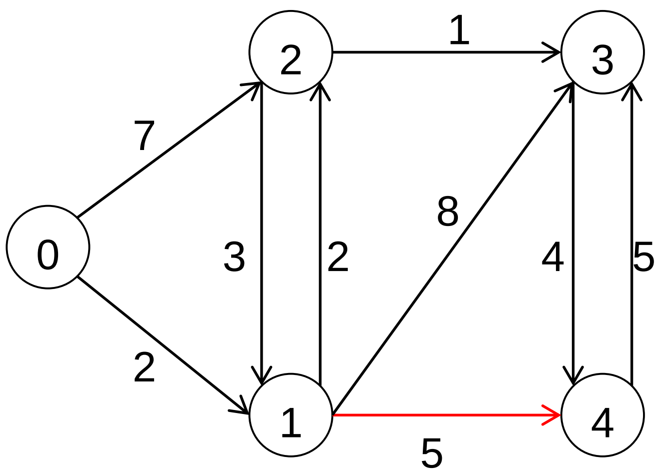
v	**0**	**1**	**2**	**3**	**4**
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0
i = ~~0~~ ~~1~~ 2
u = 3
v = 4

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
FIM
```



dist[4] > dist[1] + w(1, 4) ?
7 > 2 + 5
Não!

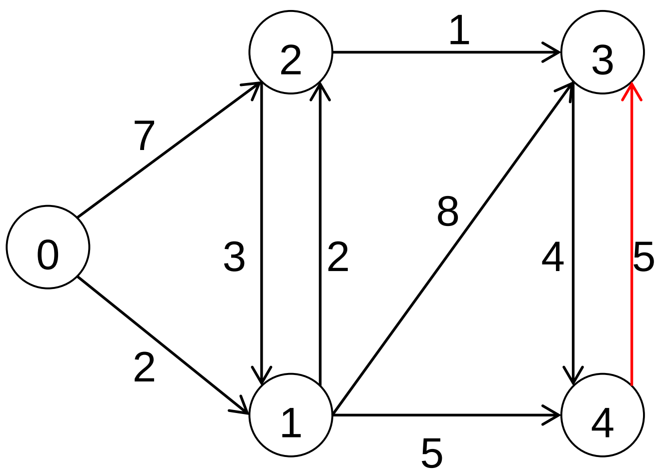
v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0
i = ~~0~~ ~~1~~ 2
u = 1
v = 4

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
FIM
```



dist[3] > dist[4] + w(4, 3) ?

5 > 7 + 5

Não!

v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0

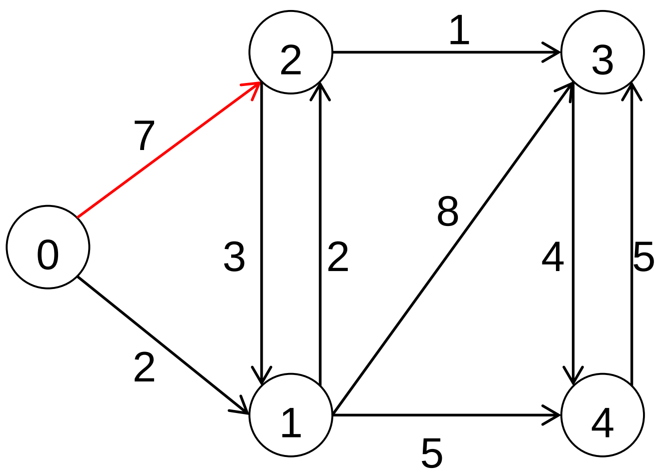
i = ~~0~~ ~~1~~ 2

u = 4

v = 3

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford




```
BELLMAN-FORD(G(V, E, w), s)
para cada vértice v em V faça
    dist [v] ← ∞
    pred[v] ← null
fim-para
dist[s] ← 0
para cada vértice i em V faça
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            dist[v] ← dist[u] + w(u, v)
            pred[v] ← u
        fim-se
    fim-para
fim-para
para cada aresta (u, v) em E faça
    se dist[v] > dist[u] + w(u, v) então
        retorne FALSE
    fim-se
fim-para
retorne TRUE
FIM
```

dist[2] > dist[0] + w(0, 2) ?

4 > 0 + 7

Não!

v	0	1	2	3	4
dist[v]	0	2	4	8	7
pred[v]	-	0	1	2	1

s = 0

i = ~~0~~ ~~1~~ ~~2~~ 3

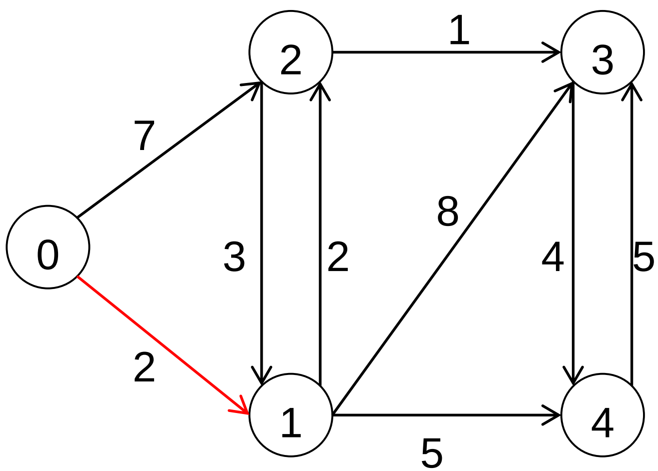
u = 0

v = 2

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
FIM
```



dist[1] > dist[0] + w(0, 1) ?

2 > 0 + 2

Não!

v	0	1	2	3	4
dist[v]	0	2	4	8	7
pred[v]	-	0	1	2	1

s = 0

i = ~~0~~ ~~1~~ ~~2~~ 3

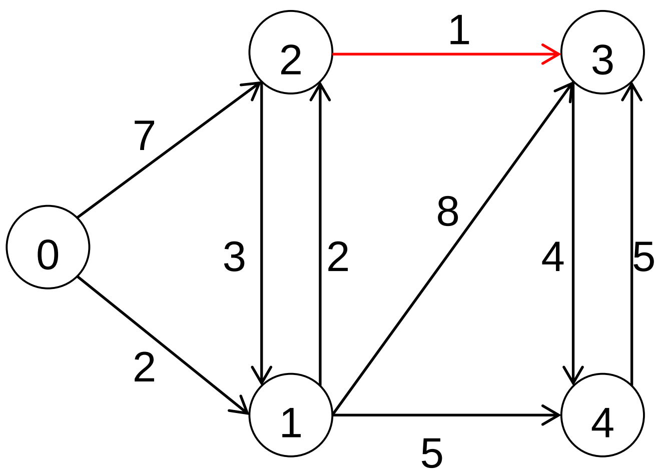
u = 0

v = 1

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
FIM
```



dist[3] > dist[2] + w(2, 3) ?

5 > 4 + 1

Não!

v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0

i = ~~0~~ ~~1~~ ~~2~~ 3

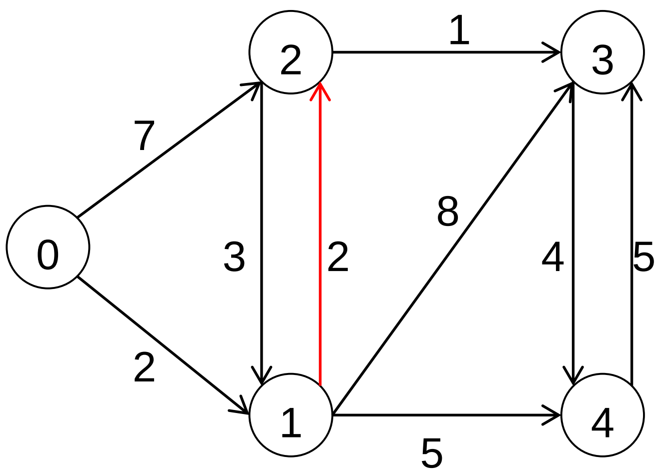
u = 2

v = 3

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
FIM
```



dist[2] > dist[1] + w(1, 2) ?
4 > 2 + 2
Não!

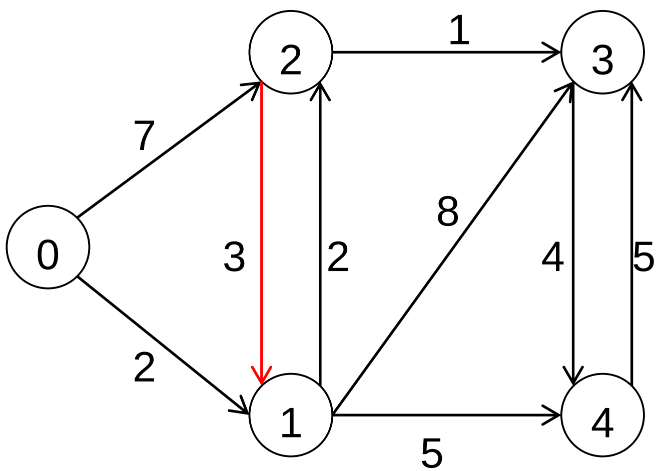
v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0
i = ~~0~~ ~~1~~ ~~2~~ 3
u = 1
v = 2

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
Fim
```



dist[1] > dist[2] + w(2, 1) ?
2 > 4 + 3
Não!

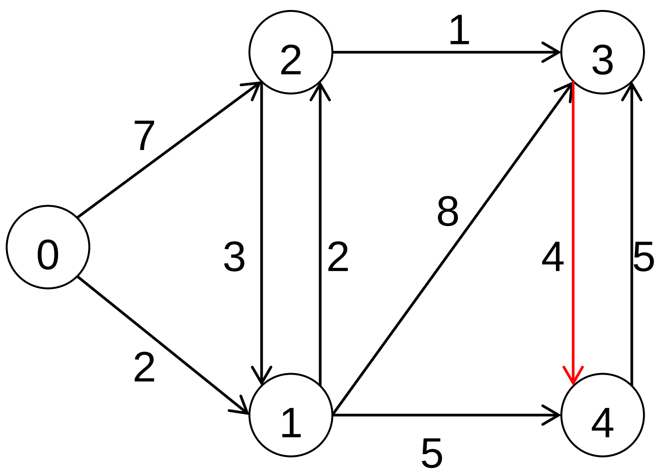
v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0
i = ~~0~~ ~~1~~ ~~2~~ 3
u = 2
v = 1

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
FIM
```



dist[4] > dist[3] + w(3, 4) ?

7 > 5 + 4

Não!

v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0

i = ~~0~~ ~~1~~ ~~2~~ 3

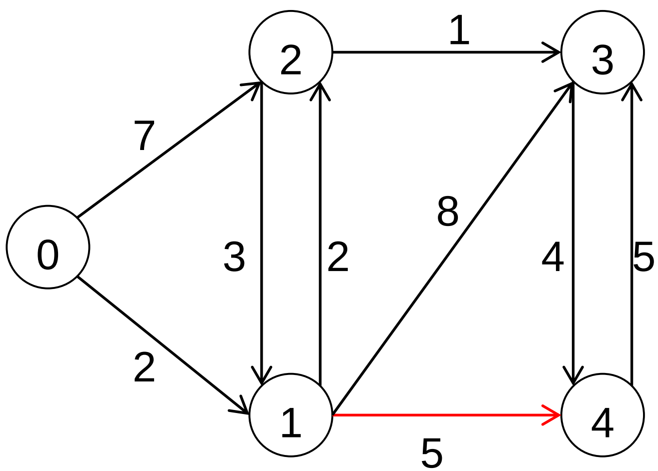
u = 3

v = 4

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
FIM
```



dist[4] > dist[1] + w(1, 4) ?

7 > 2 + 5

Não!

v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0

i = ~~0~~ ~~1~~ ~~2~~ 3

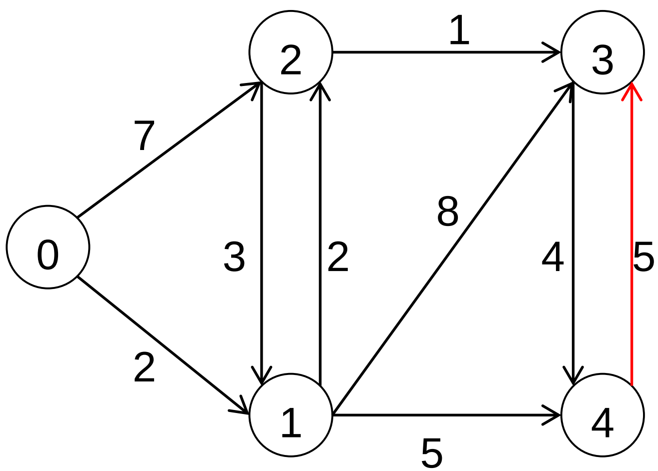
u = 1

v = 4

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
Fim
```



dist[3] > dist[4] + w(4, 3) ?

5 > 7 + 5

Não!

v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0

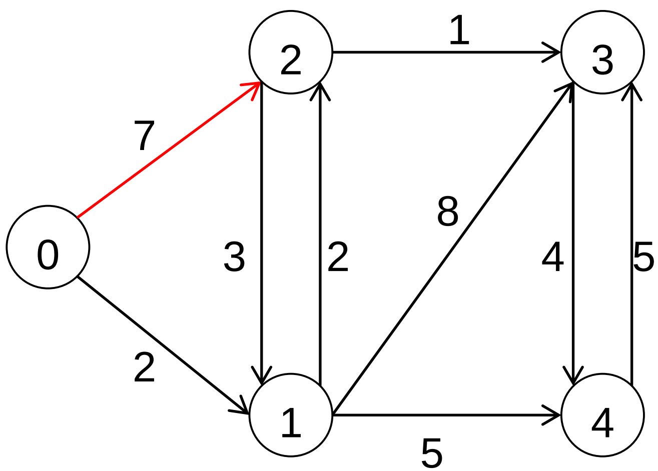
i = ~~0~~ ~~1~~ ~~2~~ 3

u = 4

v = 3

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford




```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
Fim
```

dist[2] > dist[0] + w(0, 2) ?

4 > 0 + 7

Não!

v	0	1	2	3	4
dist[v]	0	2	4	8	7
pred[v]	-	0	1	2	1

s = 0

i = ~~0~~ ~~1~~ ~~2~~ ~~3~~ 4

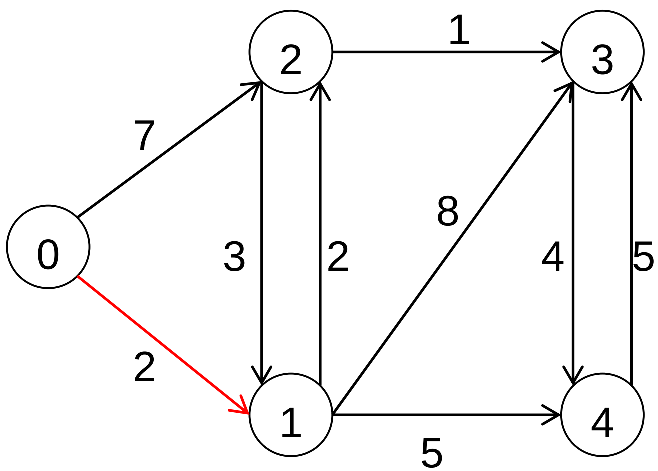
u = 0

v = 2

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
FIM
```



dist[1] > dist[0] + w(0, 1) ?

2 > 0 + 2

Não!

v	0	1	2	3	4
dist[v]	0	2	4	8	7
pred[v]	-	0	1	2	1

s = 0

i = ~~0~~ ~~1~~ ~~2~~ ~~3~~ 4

u = 0

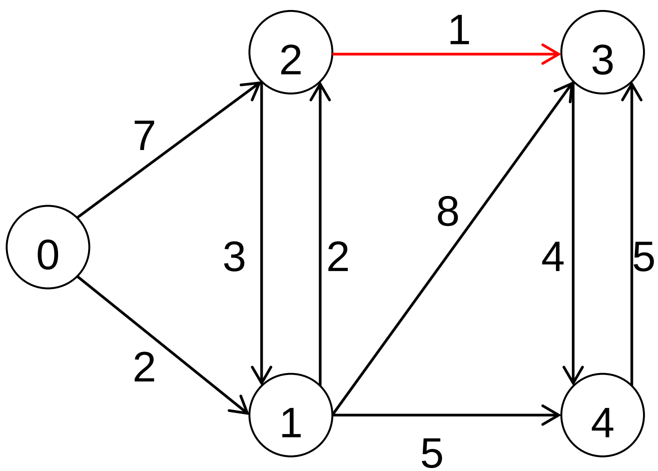
v = 1

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

BELLMAN-FORD(G(V, E, w), s)

```
para cada vértice v em V faça
    dist [v] ← ∞
    pred[v] ← null
fim-para
dist[s] ← 0
para cada vértice i em V faça
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            dist[v] ← dist[u] + w(u, v)
            pred[v] ← u
        fim-se
    fim-para
fim-para
para cada aresta (u, v) em E faça
    se dist[v] > dist[u] + w(u, v) então
        retorne FALSE
    fim-se
fim-para
retorne TRUE
Fim
```



dist[3] > dist[2] + w(2, 3) ?

5 > 4 + 1

Não!

v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0

i = ~~0~~ ~~1~~ ~~2~~ ~~3~~ 4

u = 2

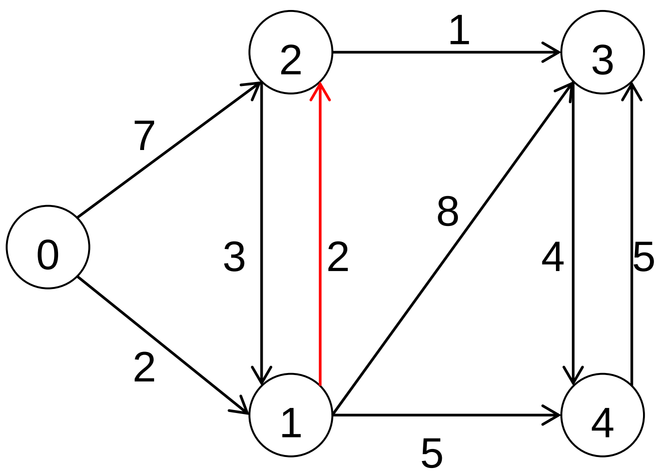
v = 3

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

BELLMAN-FORD(G(V, E, w), s)

```
para cada vértice v em V faça
    dist [v] ← ∞
    pred[v] ← null
fim-para
dist[s] ← 0
para cada vértice i em V faça
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            dist[v] ← dist[u] + w(u, v)
            pred[v] ← u
        fim-se
    fim-para
fim-para
para cada aresta (u, v) em E faça
    se dist[v] > dist[u] + w(u, v) então
        retorne FALSE
    fim-se
fim-para
retorne TRUE
FIM
```



dist[2] > dist[1] + w(1, 2) ?

4 > 2 + 2

Não!

v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0

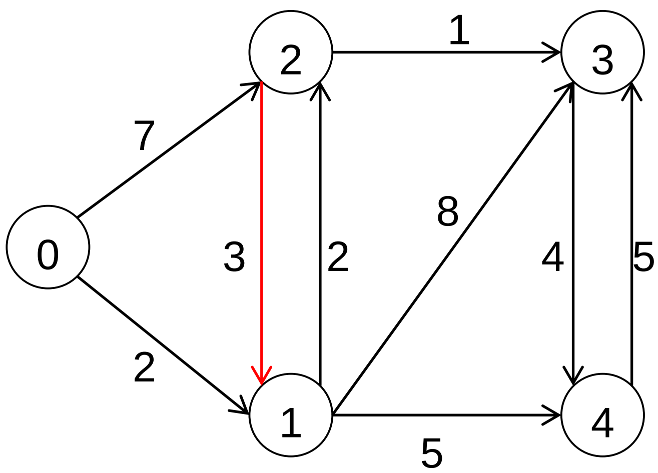
i = ~~0~~ ~~1~~ ~~2~~ ~~3~~ 4

u = 1

v = 2

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford




```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
FIM
```

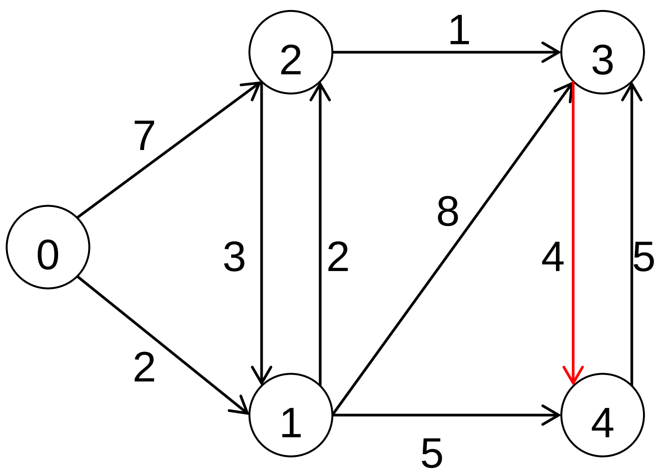
dist[1] > dist[2] + w(2, 1) ?
2 > 4 + 3
Não!

v	**0**	**1**	**2**	**3**	**4**
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0
i = ~~0~~ ~~1~~ ~~2~~ ~~3~~ 4
u = 2
v = 1

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford



BELLMAN-FORD(G(V, E, w), s)

```
para cada vértice v em V faça
    dist [v] ← ∞
    pred[v] ← null
fim-para
dist[s] ← 0
para cada vértice i em V faça
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            dist[v] ← dist[u] + w(u, v)
            pred[v] ← u
        fim-se
    fim-para
fim-para
para cada aresta (u, v) em E faça
    se dist[v] > dist[u] + w(u, v) então
        retorne FALSE
    fim-se
fim-para
retorne TRUE
FIM
```

dist[4] > dist[3] + w(3, 4) ?
7 > 5 + 4
Não!

v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

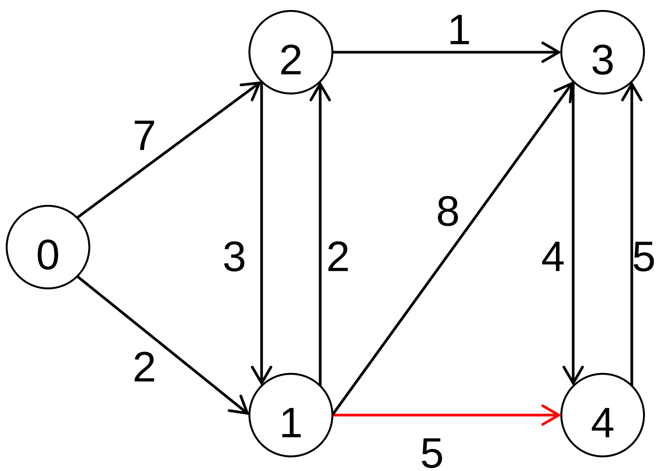
s = 0
i = $\cancel{0}$ $\cancel{1}$ $\cancel{2}$ $\cancel{3}$ 4
u = 3
v = 4

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

BELLMAN-FORD(G(V, E, w), s)

```
para cada vértice v em V faça
    dist [v] ← ∞
    pred[v] ← null
fim-para
dist[s] ← 0
para cada vértice i em V faça
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            dist[v] ← dist[u] + w(u, v)
            pred[v] ← u
        fim-se
    fim-para
fim-para
para cada aresta (u, v) em E faça
    se dist[v] > dist[u] + w(u, v) então
        retorne FALSE
    fim-se
fim-para
retorne TRUE
FIM
```



dist[4] > dist[1] + w(1, 4) ?

7 > 2 + 5

Não!

v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0

i = ~~0~~ ~~1~~ ~~2~~ ~~3~~ 4

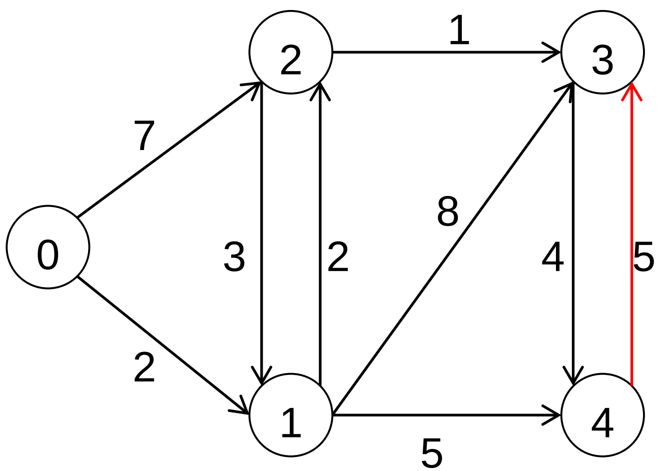
u = 1

v = 4

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
FIM
```



dist[3] > dist[4] + w(4, 3) ?

5 > 7 + 5

Não!

v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0

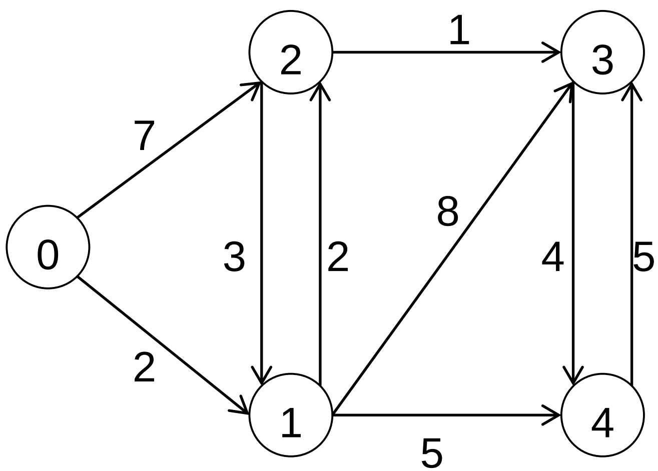
i = ~~0~~ ~~1~~ ~~2~~ ~~3~~ 4

u = 4

v = 3

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford




```
BELLMAN-FORD(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    para cada vértice i em V faça
        para cada aresta (u, v) em E faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-para
    para cada aresta (u, v) em E faça
        se dist[v] > dist[u] + w(u, v) então
            retorne FALSE
        fim-se
    fim-para
    retorne TRUE
FIM
```

v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0

i = ~~0~~ ~~1~~ ~~2~~ ~~3~~ 4

Fim do algoritmo!

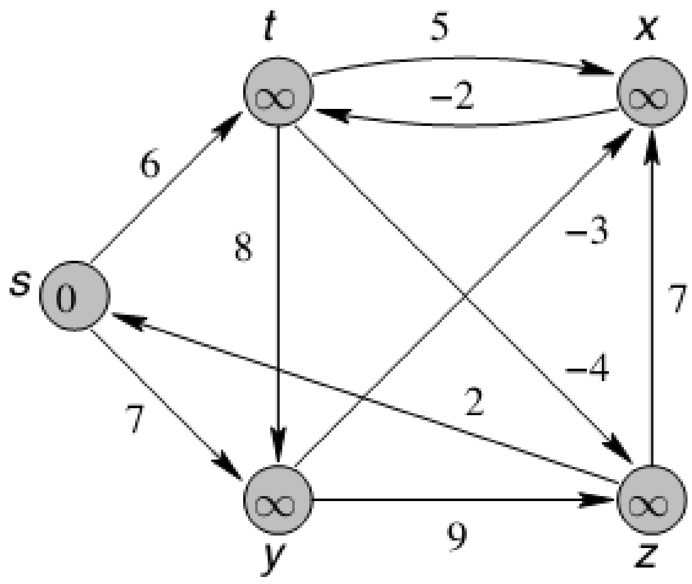
Retorne TRUE (não há ciclo de custo negativo)

E = {(0, 2), (0, 1), (2, 3), (1, 2), (1, 3), (2, 1), (3, 4), (1, 4), (4, 3)}

Algoritmo de Bellman-Ford

Exercício

- Simule a execução do algoritmo de Bellman-Ford para o grafo abaixo considerando a ordem de arestas dada



Ordem:
$(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y).$

Algoritmo de Floyd-Warshall

- O algoritmo Floyd determina as distâncias dos menores caminhos entre todos os pares de vértices de um grafo

- Funciona com arcos com peso negativo (ao contrario do Djikstra)

- Ideia: verificar todas as possíveis desigualdades triangulares no grafo!

Algoritmo de Floyd-Warshall

```
FLOYD-WARSHALL(G(V, E, w))   //Parâmetros: representação de grafo (V, E, w)
    para cada vértice i em V faça
        para cada vértice j em V faça
            se i = j então
                dist [ i ][ j ] ← 0
            senão se (i, j) ∊ E então      //Existe aresta entre i e j
                dist [ i ][ j ] ← w[ i ][ j ]
                pred [ i ] [ j ] ← i
            senão
                dist [ i ][ j ] ← ∞
                pred [ i ] [ j ] ← null
            fim-se
        fim-para
    fim-para
    para cada vértice k em V faça
        para cada vértice i em V faça
            para cada vértice j em V faça
                se dist [ i ][ j ] > dist [ i ][ k ] + dist [ k ][ j ] então
                    dist [ i ][ j ] ← dist [ i ][ k ] + dist [ k ][ j ]
                    pred [ i ][ j ] ← pred [ k ][ j ]
                fim-se
            fim-para
        fim-para
    fim-para
FIM
```

//dist: matriz que armazena a distância de cada vértice (linha) a cada vértice (coluna)

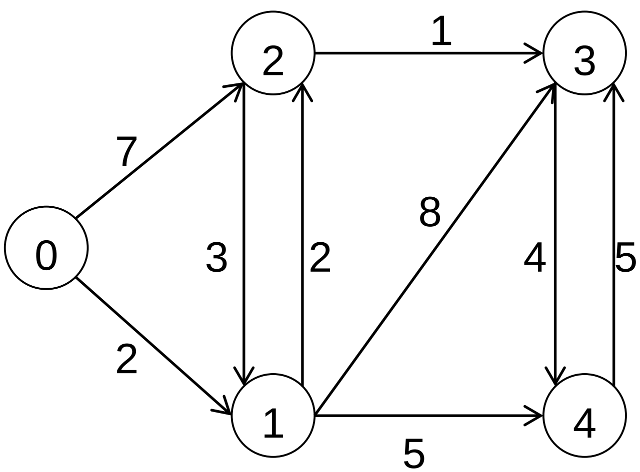
//pred: matriz que indica o predecessor de cada vértice (coluna) no caminho mínimo a partir de cada vértice (linha)

Algoritmo de Floyd-Warshall

```
FLOYD-WARSHALL(G(V, E, w))
    para cada vértice i em V faça
        para cada vértice j em V faça
            se i = j então
                dist [ i ][ j ] ← 0
            senão se (i, j) ∊ E então
                dist [ i ][ j ] ← w[ i ][ j ]
                pred [ i ] [ j ] ← i
            senão
                dist [ i ][ j ] ← ∞
                pred [ i ] [ j ] ← null
            fim-se
        fim-para
    fim-para
    para cada vértice k em V faça
        para cada vértice i em V faça
            para cada vértice j em V faça
                se dist [ i ][ j ] > dist [ i ][ k ] + dist [ k ][ j ] então
                    dist [ i ][ j ] ← dist [ i ][ k ] + dist [ k ][ j ]
                    pred [ i ][ j ] ← pred [ k ][ j ]
                fim-se
            fim-para
        fim-para
    fim-para
FIM
```

Complexidade de tempo $O(|V|^3)$

Algoritmo de Floyd-Warshall

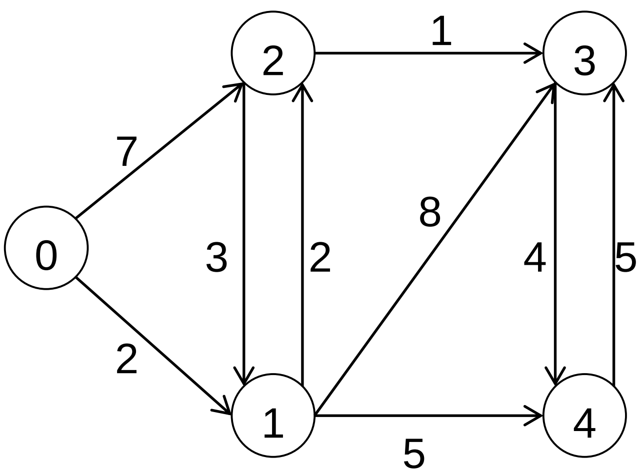



```
FLOYD-WARSHALL(G(V, E, w))
    para cada vértice i em V faça
        para cada vértice j em V faça
            se i = j então
                dist [ i ][ j ] ← 0
            senão se (i, j) ∊ E então
                dist [ i ][ j ] ← w[ i ][ j ]
                pred [ i ] [ j ] ← i
            senão
                dist [ i ][ j ] ← ∞
                pred [ i ] [ j ] ← null
            fim-se
        fim-para
    fim-para
    para cada vértice k em V faça
        para cada vértice i em V faça
            para cada vértice j em V faça
                se dist [ i ][ j ] > dist [ i ][ k ] + dist [ k ][ j ] então
                    dist [ i ][ j ] ← dist [ i ][ k ] + dist [ k ][ j ]
                    pred [ i ][ j ] ← pred [ k ][ j ]
                fim-se
            fim-para
        fim-para
    fim-para
FIM
```

dist[ ] [ ]	0	1	2	3	4
0	0	2	7	∞	∞
1	∞	0	2	8	5
2	∞	3	0	1	∞
3	∞	∞	∞	0	4
4	∞	∞	∞	5	0

pred[ ] [ ]	0	1	2	3	4
0	-	0	0	-	-
1	-	-	1	1	1
2	-	2	-	2	-
3	-	-	-	-	3
4	-	-	-	4	-

Algoritmo de Floyd-Warshall




```
FLOYD-WARSHALL(G(V, E, w))
    para cada vértice i em V faça
        para cada vértice j em V faça
            se i = j então
                dist [ i ][ j ] ← 0
            senão se (i, j) ∊ E então
                dist [ i ][ j ] ← w[ i ][ j ]
                pred [ i ] [ j ] ← i
            senão
                dist [ i ][ j ] ← ∞
                pred [ i ] [ j ] ← null
            fim-se
        fim-para
    fim-para
    para cada vértice k em V faça
        para cada vértice i em V faça
            para cada vértice j em V faça
                se dist [ i ][ j ] > dist [ i ][ k ] + dist [ k ][ j ] então
                    dist [ i ][ j ] ← dist [ i ][ k ] + dist [ k ][ j ]
                    pred [ i ][ j ] ← pred [ k ][ j ]
                fim-se
            fim-para
        fim-para
    fim-para
FIM
```

Para k = 0, fazer todas as computações

i = {0, ..., 4}

j = {0, ..., 4}

dist[ ] [ ]	**0**	**1**	**2**	**3**	**4**
0	0	2	7	∞	∞
1	∞	0	2	8	5
2	∞	3	0	1	∞
3	∞	∞	∞	0	4
4	∞	∞	∞	5	0

pred[ ] [ ]	**0**	**1**	**2**	**3**	**4**
0	-	0	0	-	-
1	-	-	1	1	1
2	-	2	-	2	-
3	-	-	-	-	3
4	-	-	-	4	-

Algoritmo de Floyd-Warshall

k = 0
i = {0, ..., 4}
j = {0, ..., 4}

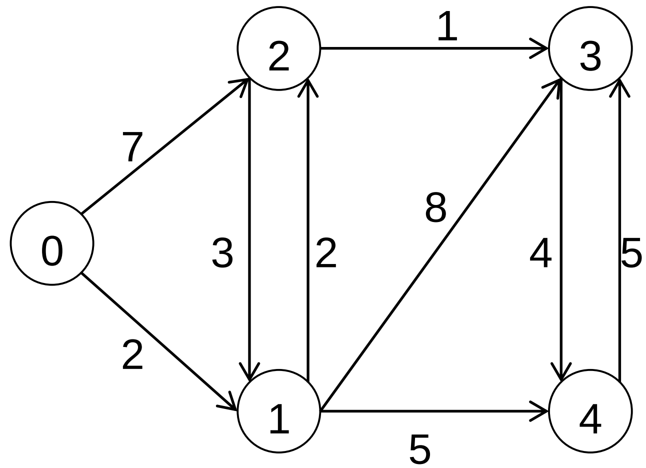
d[0][0] > d[0][0] + d[0][0] ≡ 0 > 0 + 0
d[0][1] > d[0][0] + d[0][1] ≡ 2 > 0 + 2
d[0][2] > d[0][0] + d[0][2] ≡ 7 > 0 + 7
d[0][3] > d[0][0] + d[0][3] ≡ ∞ > 0 + ∞
d[0][4] > d[0][0] + d[0][4] ≡ ∞ > 0 + ∞

d[1][0] > d[1][0] + d[0][0] ≡ ∞ > ∞ + 0
d[1][1] > d[1][0] + d[0][1] ≡ 0 > ∞ + 2
d[1][2] > d[1][0] + d[0][2] ≡ 2 > ∞ + 7
d[1][3] > d[1][0] + d[0][3] ≡ 8 > ∞ + ∞
d[1][4] > d[1][0] + d[0][4] ≡ 5 > ∞ + ∞

d[2][0] > d[2][0] + d[0][0] ≡ ∞ > ∞ + 0
d[2][1] > d[2][0] + d[0][1] ≡ 3 > ∞ + 2
d[2][2] > d[2][0] + d[0][2] ≡ 0 > ∞ + 7
d[2][3] > d[2][0] + d[0][3] ≡ 1 > ∞ + ∞
d[2][4] > d[2][0] + d[0][4] ≡ ∞ > ∞ + ∞

d[3][0] > d[3][0] + d[0][0] ≡ ∞ > ∞ + 0
d[3][1] > d[3][0] + d[0][1] ≡ ∞ > ∞ + 2
d[3][2] > d[3][0] + d[0][2] ≡ ∞ > ∞ + 7
d[3][3] > d[3][0] + d[0][3] ≡ 0 > ∞ + ∞
d[3][4] > d[3][0] + d[0][4] ≡ 4> ∞ + ∞

d[4][0] > d[4][0] + d[0][0] ≡ ∞ > ∞ + 0
d[4][1] > d[4][0] + d[0][1] ≡ ∞ > ∞ + 2
d[4][2] > d[4][0] + d[0][2] ≡ ∞ > ∞ + 7
d[4][3] > d[4][0] + d[0][3] ≡ 5 > ∞ + ∞
d[4][4] > d[4][0] + d[0][4] ≡ 0 > ∞ + ∞



dist[ ] [ ]	**0**	**1**	**2**	**3**	**4**
0	0	2	7	∞	∞
1	∞	0	2	8	5
2	∞	3	0	1	∞
3	∞	∞	∞	0	4
4	∞	∞	∞	5	0

pred[ ] [ ]	**0**	**1**	**2**	**3**	**4**
0	-	0	0	-	-
1	-	-	1	1	1
2	-	2	-	2	-
3	-	-	-	-	3
4	-	-	-	4	-

Algoritmo de Floyd-Warshall

k = 1
i = {0, ..., 4}
j = {0, ..., 4}

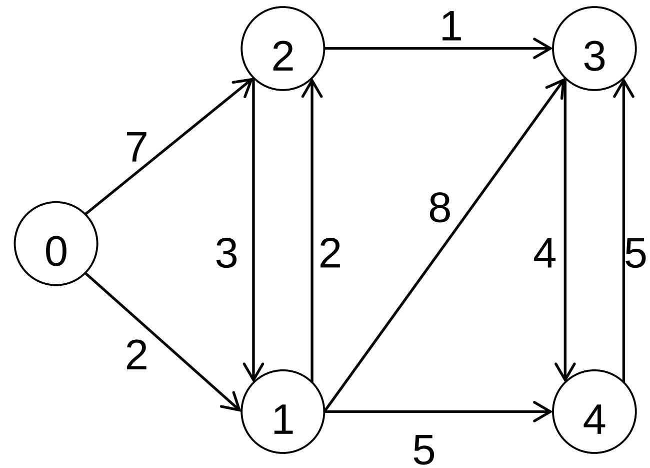
d[0][0] > d[0][1] + d[1][0] ≡ 0 > 2 + ∞
d[0][1] > d[0][1] + d[1][1] ≡ 2 > 2 + 0
d[0][2] > d[0][1] + d[1][2] ≡ **7 > 2 + 2**
d[0][3] > d[0][1] + d[1][3] ≡ **∞ > 2 + 8**
d[0][4] > d[0][1] + d[1][4] ≡ **∞ > 2 + 5**

d[1][0] > d[1][1] + d[1][0] ≡ ∞ > 0 + ∞
d[1][1] > d[1][1] + d[1][1] ≡ 0 > 0 + 0
d[1][2] > d[1][1] + d[1][2] ≡ 2 > 0 + 2
d[1][3] > d[1][1] + d[1][3] ≡ 8 > 0 + 8
d[1][4] > d[1][1] + d[1][4] ≡ 5 > 0 + 5

d[2][0] > d[2][1] + d[1][0] ≡ ∞ > 3 + ∞
d[2][1] > d[2][1] + d[1][1] ≡ 3 > 3 + 0
d[2][2] > d[2][1] + d[1][2] ≡ 0 > 3 + 2
d[2][3] > d[2][1] + d[1][3] ≡ 1 > 3 + 8
d[2][4] > d[2][1] + d[1][4] ≡ **∞ > 3 + 5**

d[3][0] > d[3][1] + d[1][0] ≡ ∞ > ∞ + ∞
d[3][1] > d[3][1] + d[1][1] ≡ ∞ > ∞ + 0
d[3][2] > d[3][1] + d[1][2] ≡ ∞ > ∞ + 2
d[3][3] > d[3][1] + d[1][3] ≡ 0 > ∞ + 8
d[3][4] > d[3][1] + d[1][4] ≡ 4 > ∞ + 5

d[4][0] > d[4][1] + d[1][0] ≡ ∞ > ∞ + ∞
d[4][1] > d[4][1] + d[1][1] ≡ ∞ > ∞ + 0
d[4][2] > d[4][1] + d[1][2] ≡ ∞ > ∞ + 2
d[4][3] > d[4][1] + d[1][3] ≡ 5 > ∞ + 8
d[4][4] > d[4][1] + d[1][4] ≡ 0 > ∞ + 5



dist[ ] [ ]	**0**	**1**	**2**	**3**	**4**
0	0	2	~~7~~ 4	~~∞~~ 10	~~∞~~ 7
1	∞	0	2	8	5
2	∞	3	0	1	~~∞~~ 8
3	∞	∞	∞	0	4
4	∞	∞	∞	5	0

pred[ ] [ ]	**0**	**1**	**2**	**3**	**4**
0	-	0	~~0~~ 1	- 1	- 1
1	-	-	1	1	1
2	-	2	-	2	- 1
3	-	-	-	-	3
4	-	-	-	4	-

Algoritmo de Floyd-Warshall

k = 2
i = {0, ..., 4}
j = {0, ..., 4}

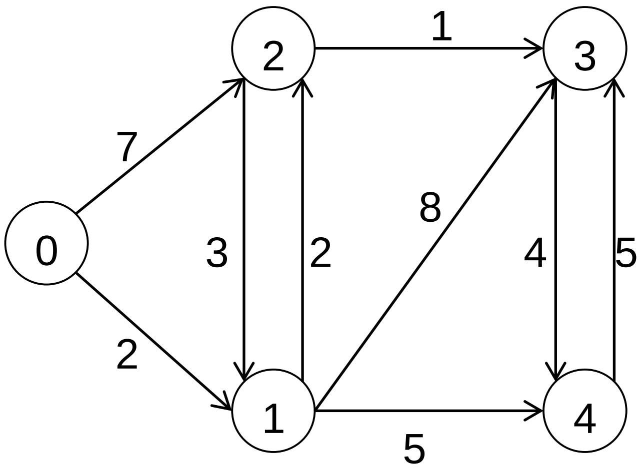
d[0][0] > d[0][2] + d[2][0] ≡ 0 > 4 + ∞
d[0][1] > d[0][2] + d[2][1] ≡ 2 > 4 + 3
d[0][2] > d[0][2] + d[2][2] ≡ 4 > 4 + 0
d[0][3] > d[0][2] + d[2][3] ≡ **10 > 4 + 1**
d[0][4] > d[0][2] + d[2][4] ≡ 7 > 4 + 8

d[1][0] > d[1][2] + d[2][0] ≡ ∞ > 2 + ∞
d[1][1] > d[1][2] + d[2][1] ≡ 0 > 2 + 3
d[1][2] > d[1][2] + d[2][2] ≡ 2 > 2 + 0
d[1][3] > d[1][2] + d[2][3] ≡ **8 > 2 + 1**
d[1][4] > d[1][2] + d[2][4] ≡ 5 > 2 + 8

d[2][0] > d[2][2] + d[2][0] ≡ ∞ > 0 + ∞
d[2][1] > d[2][2] + d[2][1] ≡ 3 > 0 + 3
d[2][2] > d[2][2] + d[2][2] ≡ 0 > 0 + 0
d[2][3] > d[2][2] + d[2][3] ≡ 1 > 0 + 1
d[2][4] > d[2][2] + d[2][4] ≡ 8 > 0 + 8

d[3][0] > d[3][2] + d[2][0] ≡ ∞ > ∞ + ∞
d[3][1] > d[3][2] + d[2][1] ≡ ∞ > ∞ + 3
d[3][2] > d[3][2] + d[2][2] ≡ ∞ > ∞ + 0
d[3][3] > d[3][2] + d[2][3] ≡ 0 > ∞ + 1
d[3][4] > d[3][2] + d[2][4] ≡ 4 > ∞ + 8

d[4][0] > d[4][2] + d[2][0] ≡ ∞ > ∞ + ∞
d[4][1] > d[4][2] + d[2][1] ≡ ∞ > ∞ + 3
d[4][2] > d[4][2] + d[2][2] ≡ ∞ > ∞ + 0
d[4][3] > d[4][2] + d[2][3] ≡ 5 > ∞ + 1
d[4][4] > d[4][2] + d[2][4] ≡ 0 > ∞ + 8



dist[ ] [ ]	**0**	**1**	**2**	**3**	**4**
0	0	2	4	~~10~~ 5	7
1	∞	0	2	~~8~~ 3	5
2	∞	3	0	1	8
3	∞	∞	∞	0	4
4	∞	∞	∞	5	0

pred[ ] [ ]	**0**	**1**	**2**	**3**	**4**
0	-	0	1	~~1~~ 2	1
1	-	-	1	~~1~~ 2	1
2	-	2	-	2	1
3	-	-	-	-	3
4	-	-	-	4	-

Algoritmo de Floyd-Warshall

k = 3
i = {0, ..., 4}
j = {0, ..., 4}

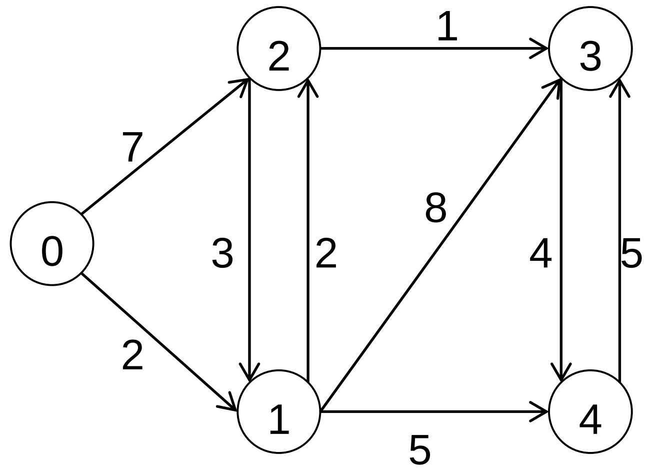
d[0][0] > d[0][3] + d[3][0] ≡ 0 > 5 + ∞
d[0][1] > d[0][3] + d[3][1] ≡ 2 > 5 + ∞
d[0][2] > d[0][3] + d[3][2] ≡ 4 > 5 + ∞
d[0][3] > d[0][3] + d[3][3] ≡ 5 > 5 + 0
d[0][4] > d[0][3] + d[3][4] ≡ 7 > 5 + 4

d[1][0] > d[1][3] + d[3][0] ≡ ∞ > 3 + ∞
d[1][1] > d[1][3] + d[3][1] ≡ 0 > 3 + ∞
d[1][2] > d[1][3] + d[3][2] ≡ 2 > 3 + ∞
d[1][3] > d[1][3] + d[3][3] ≡ 3 > 3 + 0
d[1][4] > d[1][3] + d[3][4] ≡ 5 > 3 + 4

d[2][0] > d[2][3] + d[3][0] ≡ ∞ > 1 + ∞
d[2][1] > d[2][3] + d[3][1] ≡ 3 > 1 + ∞
d[2][2] > d[2][3] + d[3][2] ≡ 0 > 1 + ∞
d[2][3] > d[2][3] + d[3][3] ≡ 1 > 1 + 0
d[2][4] > d[2][3] + d[3][4] ≡ **8 > 1 + 4**

d[3][0] > d[3][3] + d[3][0] ≡ ∞ > 0 + ∞
d[3][1] > d[3][3] + d[3][1] ≡ ∞ > 0 + ∞
d[3][2] > d[3][3] + d[3][2] ≡ ∞ > 0 + ∞
d[3][3] > d[3][3] + d[3][3] ≡ 0 > 0 + 0
d[3][4] > d[3][3] + d[3][4] ≡ 4 > 0 + 4

d[4][0] > d[4][3] + d[3][0] ≡ ∞ > 5 + ∞
d[4][1] > d[4][3] + d[3][1] ≡ ∞ > 5 + ∞
d[4][2] > d[4][3] + d[3][2] ≡ ∞ > 5 + ∞
d[4][3] > d[4][3] + d[3][3] ≡ 5 > 5 + 0
d[4][4] > d[4][3] + d[3][4] ≡ 0 > 5 + 4



dist[ ] [ ]	**0**	**1**	**2**	**3**	**4**
0	0	2	4	5	7
1	∞	0	2	3	5
2	∞	3	0	1	~~8~~ 5
3	∞	∞	∞	0	4
4	∞	∞	∞	5	0

pred[ ] [ ]	**0**	**1**	**2**	**3**	**4**
0	-	0	1	2	1
1	-	-	1	2	1
2	-	2	-	2	~~1~~ 3
3	-	-	-	-	3
4	-	-	-	4	-

Algoritmo de Floyd-Warshall

k = 4
i = {0, ..., 4}
j = {0, ..., 4}

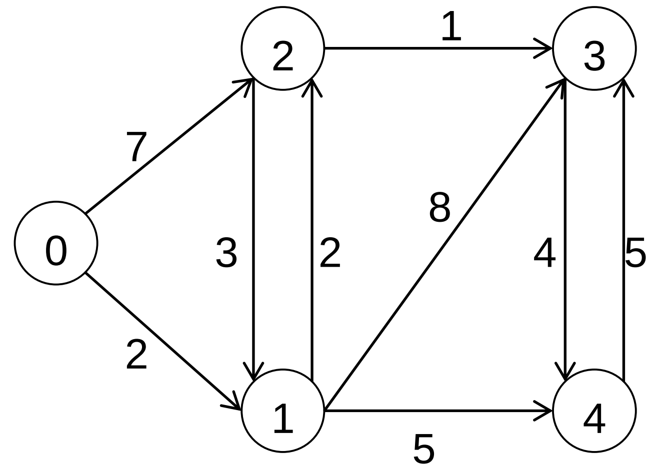
d[0][0] > d[0][4] + d[4][0] ≡ 0 > 7 + ∞
d[0][1] > d[0][4] + d[4][1] ≡ 2 > 7 + ∞
d[0][2] > d[0][4] + d[4][2] ≡ 4 > 7 + ∞
d[0][3] > d[0][4] + d[4][3] ≡ 5 > 7 + 5
d[0][4] > d[0][4] + d[4][4] ≡ 7 > 7 + 0

d[1][0] > d[1][4] + d[4][0] ≡ ∞ > 5 + ∞
d[1][1] > d[1][4] + d[4][1] ≡ 0 > 5 + ∞
d[1][2] > d[1][4] + d[4][2] ≡ 2 > 5 + ∞
d[1][3] > d[1][4] + d[4][3] ≡ 3 > 5 + 5
d[1][4] > d[1][4] + d[4][4] ≡ 5 > 5 + 0

d[2][0] > d[2][4] + d[4][0] ≡ ∞ > 5 + ∞
d[2][1] > d[2][4] + d[4][1] ≡ 3 > 5 + ∞
d[2][2] > d[2][4] + d[4][2] ≡ 0 > 5 + ∞
d[2][3] > d[2][4] + d[4][3] ≡ 1 > 5 + 5
d[2][4] > d[2][4] + d[4][4] ≡ 5 > 5 + 0

d[3][0] > d[3][4] + d[4][0] ≡ ∞ > 4 + ∞
d[3][1] > d[3][4] + d[4][1] ≡ ∞ > 4 + ∞
d[3][2] > d[3][4] + d[4][2] ≡ ∞ > 4 + ∞
d[3][3] > d[3][4] + d[4][3] ≡ 0 > 4 + 5
d[3][4] > d[3][4] + d[4][4] ≡ 4 > 4 + 0

d[4][0] > d[4][4] + d[4][0] ≡ ∞ > 0 + ∞
d[4][1] > d[4][4] + d[4][1] ≡ ∞ > 0 + ∞
d[4][2] > d[4][4] + d[4][2] ≡ ∞ > 0 + ∞
d[4][3] > d[4][4] + d[4][3] ≡ 5 > 0 + 5
d[4][4] > d[4][4] + d[4][4] ≡ 0 > 0 + 0



dist[ ] [ ]	**0**	**1**	**2**	**3**	**4**
0	0	2	4	5	7
1	∞	0	2	3	5
2	∞	3	0	1	5
3	∞	∞	∞	0	4
4	∞	∞	∞	5	0

pred[ ] [ ]	**0**	**1**	**2**	**3**	**4**
0	-	0	1	2	1
1	-	-	1	2	1
2	-	2	-	2	3
3	-	-	-	-	3
4	-	-	-	4	-

Algoritmo de Floyd-Warshall

Exercício

- Proponha um grafo de três vértices, com ciclo de peso negativo, e simule a execução do algoritmo Floyd-Warshall sobre o mesmo

Recuperando o Caminho Mínimo

REC-CAMINHO(s, t, pred[ ]) : C

1. C ← {t} //C: lista que contém os vértices do caminho de s a t
2. aux ← t
3. **enquanto** aux ≠ s **faça**
4. aux ← pred[aux]
5. adicione aux ao início de C
6. **fim-enquanto**
7. **retorne C**

FIM

Revisão

- Dijkstra
  - Obter o caminho mínimo para um vértice por iteração até checar todos os vértices
  - $O(|V|^2)$ – pode ser melhorado
- Bellman-Ford
  - Avaliar aresta a aresta para progressivamente diminuir as estimativas de distância até encontrar o menor caminho
  - Detecta ciclo de custo negativo (se existir) e aceita arestas de peso negativo
  - $O(|V| \times |E|)$
- Floyd-Warshall
  - Verificar todas as possíveis desigualdades triangulares no grafo!
  - $O(|V|^3)$ – calcula o menor caminho para todos os pares de vértices

Bibliografia

- Goldbarg, M.; Goldbarg, E. **Grafos: Conceitos, algoritmos e aplicações**. 1ª edição. Elsevier, 2012.

- Boaventura Netto, P. O. **Grafos: Teoria, Modelos, Algoritmos**. 4a edição. Edgar Blucher, 2012.